PRACTICAL SYSTEMS FOR STRENGTHENING AND WEAKENING BINARY ANALYSIS FRAMEWORKS

A Dissertation
Presented to
The Academic Faculty

By

Jinho Jung

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computing
Department of Computer Science

Georgia Institute of Technology

May 2021

PRACTICAL SYSTEMS FOR STRENGTHENING AND WEAKENING BINARY ANALYSIS FRAMEWORKS

Thesis committee:

Dr. Taesoo Kim (Advisor)
School of Computer Science
Georgia Institute of Technology

Dr. Wenke Lee
School of Computer Science
Georgia Institute of Technology

Dr. Joy Arulraj (Co-advisor)
School of Computer Science
Georgia Institute of Technology

Dr. Kyu Hyung Lee
The Department of Computer Science
University of Georgia

Dr. Paul Pearce (Co-advisor)
School of Computer Science
Georgia Institute of Technology

Date approved: Apr 6, 2021

ACKNOWLEDGEMENTS

My Ph.D. thesis would not exist without the support of many people.

First of all, I would like to thank to my advisors. I was very fortunate to have wonderful advisors, Dr. Taesoo Kim, Dr. Joy Arulraj, Dr. Paul Pearce, and my hidden advisor Dr. Hong Hu. Dr. Taesoo Kim has been always "nice" to me. He taught me how to concentrate to the research topic and boost the outcome, excluding any minor or personal problems. Considering my first year of study, I would say he created something out of nothing. I am so grateful for it. Dr. Joy Arulraj is an intelligent researcher. I could not have a chance to work with him if I am not a Georgia Tech student. Collaboration with him made the most efficient performance during my study. I would like to thank to Dr. Paul Pearce and show my regret about the unfinished project BLUEPRINT. It was a great pleasure working with him for this project. Dr. Hong Hu collaborated with me for most of my research projects. Without his close support, I may spend two more years finishing my Ph.D. program.

I also would like to appreciate my dissertation committee: Dr. Wenke Lee and Dr. Kyu Hyung Lee for their comments and suggestions for my thesis. Dr. Wenke Lee guided me to conduct research for MLSPLOIT project and Dr. Kyu Hyung Lee supported AVPASS and FUZZIFICATION project.

I have been lucky to study and hack with many researchers and staffs at Georgia Tech: Dr. Changwoo Min, Dr. Kangjie Lu, Dr. YeongJin Jang, Dr. Woonhak Kang, Dr. Hyungon Moon, Dr. Sangho Lee, Dr. Chanil Jeon, Dr. Mohan Kumar, Dr. Steffen Maass, Dr. Ming-Wei Shih, Dr. Meng Xu, Dr. Daehee Jang, Dr. Sanidhya Kashyap, Dr. Hyungjoon Koo, Dr. Insu Yun, Max Wolosky, ChangSeok Oh, Wen Xu, Ren Ding, Soyeon Park, Fan Sang, Seulbae Kim, Hanqing Zhao, Yechan Bae, Sujin Park, Mansour Alharthi, Ammar Askar, Jungwon Lim, Yonghwi Jin, Yu-Fu Fu, Kevin Stevens, Stephen Tong, Trinh Doan, Elizabeth Ndongi, Sue Jean Chae, and Chulwon Kang.

Finally, I would like to thank my parents, Jungrok Choi and Haebok Jung, and my parents-in-law, Soonae Lee and Myungjoo Won for their support. Also, I appreciate to Sungmi Kim and Chin's family for being a good friend. parent. Especially, I thank to my wife, Keumyoung, and my little son, Noah. I could not finish the long journey without their support.

Noah! My little boy! You found this document. I have something for you. Find and tell me "I am ready to get it now".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Binary analysis detects software vulnerability. Cutting-edge analysis techniques can quickly and automatically explore the internals of a program and report any discovered problems. Therefore, developers commonly use various analysis techniques as part of their software development process. Unfortunately, it also means that such techniques and the automatic natures of binary analysis methods are appealing to adversaries who are looking for zero-day vulnerabilities.

In this thesis, binary analysis is considered a double-edged sword for the users, based on their purpose. To deliver the benefit of the binary analysis only for credible users such as developers or testers, this thesis aims to present a practical system to strengthening the binary analysis for the trusted parties and weakening the power of the binary analysis against the untrusted groups exclusively.

To achieve the aforementioned goals, this thesis presents the new domain of the binary analysis in two directions: 1) a protection technique against the fuzz testing and 2) a new binary analysis system to expand the applicability of the current binary analysis techniques. The mitigation approach will help developers protect the released software from attackers who can apply fuzzing techniques. On the other hand, the new binary analysis frameworks will provide a set of solutions to address the challenges that COTS binary fuzzing faces.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Binary analysis is widely adopted for detecting software bugs but brings the similar amount of benefits for both the trusted parties and the adversaries. Recently, various state-of-the-art techniques contributed to unearth the critical software vulnerabilities for the developers. Unfortunately, those advanced techniques also empowered the adversaries who are hunting for zero-day vulnerabilities.

Among the widely used binary analysis techniques, fuzzing is a software testing method that aims to find software bugs automatically. It keeps running the program with randomly generated inputs and waits for bug-exposing behaviors such as crashing or hanging. Due to its effectiveness and scalability, it has become a standard practice to detect security problems in complex, modern software [1, 2, 3, 4, 5, 6, 7]; thus developers commonly use fuzzing as part of test integration throughout the software development process.

Nevertheless, advanced fuzzing techniques can also be used by malicious attackers to find zero-day vulnerabilities. Recent studies [8, 9] confirm that attackers predominantly prefer fuzzing tools over others (*e.g.*, reverse engineering) in finding vulnerabilities. For example, a survey of information security experts [10] shows that fuzzing techniques discover 4.8 times more bugs than static analysis or manual detection. Therefore, developers might want to apply *anti-fuzzing* techniques on their products to hinder any fuzzing attempts conducted by attackers, similar in concept to using obfuscation techniques to cripple reverse engineering [11, 12]. To solve this problem, we propose a new mitigation approach, called FUZZIFICATION, that helps developers protect the released, binary-only software from attackers who are capable of applying state-of-the-art fuzzing techniques.

Besides the anti-fuzzing problem, an important challenge we are encountering in the binary analysis is Windows application fuzzing. Fuzzing on the Windows OS is not well-explored. Since existing fuzzing techniques are mainly applied to Unix-like OSes and Windows application contains various challenges for applying the fuzzing technique, few of them work as well on Windows platforms. Therefore, Windows applications are not free from bugs. A recent report shows that in the past 12 years, 70% of all security vulnerabilities on Windows systems are memory safety issues [13]. In fact, due to the dominance of the Windows operating system, its applications remain the most lucrative targets for malicious attackers [14, 15, 16, 17]. To bring popular fuzzing techniques to the Windows platform, this thesis investigates common applications and state-of-the-art fuzzers, and identifies various challenges of fuzzing applications on Windows:

1.2 Research Outline

This thesis proposes two systems to tackle the aforementioned problems. First, this thesis proposes a technique called FUZZIFICATION that consists of three anti-fuzzing techniques for developers to protect their programs from malicious fuzzing attempts: SpeedBump, BranchTrap, and AntiHybrid. The SpeedBump technique aims to slow program execution during fuzzing. It injects delays to *cold* paths, which normal executions rarely reach but that fuzzed executions frequently visit. The BranchTrap technique inserts a large number of input-sensitive jumps into the program so that any input drift will significantly change the execution path. This will induce coverage-based fuzzing tools to spend their efforts on injected bug-free paths instead of on the real ones. The AntiHybrid technique aims to thwart hybrid fuzzing approaches that incorporate traditional fuzzing methods with dynamic taint analysis and symbolic execution.

Second, this thesis presents WINNIE, an end-to-end system to address the challenges on the Windows fuzzing and make the testing more practical. WINNIE contains two components: a harness generator that automatically synthesizes harnesses from the program

binary alone, and an efficient Windows fork-server. To construct plausible harnesses, our harness generator combines both dynamic and static analysis. We run the target program against several inputs, collect execution traces, and identify interesting functions and libraries that are suitable for fuzzing. Then, our generator searches the execution traces to collect all function calls to candidate libraries, and extracts them to form a harness skeleton. Finally, we try to identify the relationships between different function calls and arguments to build a full harness.

CHAPTER 2
RELATED WORK

In this chapter, we introduce the various binary analysis techniques for testing binary such as fuzzing (§2.1), concolic execution (§2.2), and anti-fuzzing techniques (§2.3).

2.1 Fuzzing

Since the first proposal by Barton Miller in 1990 [1], fuzzing has evolved into a standard method for automatic program testing and bug finding. Various fuzzing techniques and tools have been proposed [18, 19, 20, 21, 22], developed [2, 3, 4, 5, 6, 7], and used to find a large number of program bugs [23, 2, 24, 25, 26]. There are continuous efforts to help improve fuzzing efficiency by developing a more effective feedback loop [27], proposing new OS primitives [28], and utilizing clusters for large-scale fuzzing [29, 30, 31].

Recently, researchers have been using fuzzing as a general way to explore program paths with specialties, such as maximizing CPU usage [32], reaching a particular code location [33], and verifying the deep learning result empirically [34]. All these works result in a significant improvement to software security and reliability. In this thesis, we focus on the opposite side of the double-edged sword, where attackers abuse fuzzing techniques to find zero-day vulnerabilities and thus launch a sophisticated cyber attack. We build effective methods to hinder attackers on bug finding using FUZZIFICATION, which can provide developers and trusted researchers time to defeat the adversarial fuzzing effort.

2.1.1 Coverage-Guided Fuzzing

Coverage-guided fuzzing becomes popular especially since AFL [35] has shown its effectiveness. AFL prioritizes inputs that likely reveal new paths by collecting coverage information during program execution to assess generated inputs, enabling quick coverage expansion.

Also, AFLFast [36] uses a Markov chain model to prioritize paths with low reachability, and CollAFL [37] provides accurate coverage information to mitigate path collisions.

However, fuzzing has a fundamental limitation: it cannot traverse paths beyond narrow-ranged input constraints (e.g., a magic value). To overcome such a limitation, VUzzer [38] develops application-aware mutation techniques by performing static and dynamic program analysis. Steelix [39] recovers correct magic values by collecting comparison progress information during program execution. FairFuzz [40] discovers magic values and prevents their mutations with program analysis and heuristics. Angora [41] adopts taint tracking, shape and type inference, and a gradient-descent-based search strategy to solve path constraints efficiently. These approaches, however, can only handle certain types of constraints. In contrast, WINNIE relies on symbolic execution such that it has a chance to satisfy any kinds of constraints. In addition, a recent study, T-Fuzz [42], transforms a program itself to cover more interesting code paths, which could be combined with WINNIE to remove unsolvable constraints from the program.

2.1.2 Hybrid Fuzzing

The concept of hybrid fuzzing is first proposed by Majumdar and Sen [43]. Later, Driller [44] demonstrated its effectiveness in DARPA CGC with a refined implementation. In both studies, the majority of path exploration is offloaded to the fuzzer, while concolic execution is selectively used to drive execution across the paths that are guarded by narrow-ranged constraints. Pak [45] also proposes a similar idea, but it is limited to the frontier nodes that are mainly magic value checks at early execution stages. However, these hybrid fuzzers use general concolic executors that are not only slow but also incompatible with hybrid fuzzing. On the contrary, WINNIE is tailored for hybrid fuzzing, so that it can scale to detect bugs from real-world software.

2.2 Concolic Execution

Concolic execution is a path-exploring technique that performs symbolic execution along a concrete execution path to direct the program to new execution paths. Concolic execution has been largely adopted for automatic vulnerability finding from source code [46, 47, 48] to binary [49, 50, 51, 52, 53].

However, concolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with a program size. To mitigate this problem, SAGE [51, 54] proposes generational search to maximize the number of test cases in one execution and applies unrelated constraint solving [55]. Dowser [56] uses static analysis and taint analysis to guide concolic execution and minimizes the number of symbolic expressions to find buffer overflow vulnerabilities. Mayhem [50] combines forking-based symbolic execution and re-execution-based symbolic execution to balance performance and memory usage. In contrast, WINNIE uses (1) fuzzing to explore most paths to avoid the path explosion problem, (2) generic heuristics (e.g., basic block pruning) without assuming any specific bug type, and (3) instruction-level re-execution-based symbolic execution for better performance.

2.3 Anti-fuzzing Techniques

A few studies briefly discuss the concept of anti-fuzzing [57, 58, 59, 60]. Among them, Göransson *et al*. evaluated two straightforward techniques, *i.e*., crash masking to prevent fuzzers finding crashes and fuzzer detection to hide functionality when being fuzzed [58]. However, attackers can easily detect these methods and bypass them for effective fuzzing. Our system provides a fine-grained controllable method to slow the fuzzed execution and introduces effective ways to manipulate the feedback loop to fool fuzzers. We also consider defensive mechanisms to prevent attackers from removing our anti-fuzzing techniques.

DeAFL [61] provides an way to prevent bug discovery by injecting edges that create

hash conflicts. However, our method introduces BranchTrap and saturates bitmap structure, thereby also enforces hash conflicts. Hu *et al*. proposed to hinder attacks by injecting provably (but not obviously) non-exploitable bugs to the program, called "Chaff Bugs" [60]. These bugs will confuse bug analysis tools and waste attackers' effort on exploit generation. Both chaff bugs and FUZZIFICATION techniques work on close-source programs. Differently, our techniques hinder bug finding in the first place, eliminating the chance for an attacker to analyze bugs or construct exploits. Further, both techniques may affect normal-but-rare usage of the program. However, our methods, at most, introduce slow down to the execution, while improper chaff bugs lead to crashes, thus harming the usability.

CHAPTER 3
FUZZIFICATION: ANTI-FUZZING TECHNIQUES

3.1 Introduction

Fuzzing is a software testing technique that aims to find software bugs automatically. It keeps running the program with randomly generated inputs and waits for bug-exposing behaviors such as crashing or hanging. It has become a standard practice to detect security problems in complex, modern software [1, 2, 3, 4, 5, 6, 7]. Recent research has built several efficient fuzzing tools [18, 19, 20, 22, 27, 28] and found a large number of security vulnerabilities [23, 2, 24, 25, 26].

Unfortunately, advanced fuzzing techniques can also be used by malicious attackers to find zero-day vulnerabilities. Recent studies [8, 9] confirm that attackers predominantly prefer fuzzing tools over others (*e.g.*, reverse engineering) in finding vulnerabilities. For example, a survey of information security experts [10] shows that fuzzing techniques discover 4.83 times more bugs than static analysis or manual detection. Therefore, developers might want to apply *anti-fuzzing* techniques on their products to hinder fuzzing attempts by attackers, similar in concept to using obfuscation techniques to cripple reverse engineering [11, 12].

In this thesis, we propose a new direction of binary protection, called FUZZIFICATION, that hinders attackers from effectively finding bugs. Specifically, attackers may still be able to find bugs from the binary protected by FUZZIFICATION, but with significantly more effort (*e.g.*, CPU, memory, and time). Thus, developers or other trusted parties who get the original binary are able to detect program bugs and synthesize patches before attackers widely abuse them. An effective FUZZIFICATION technique should enable the following three features. First, it should be effective for hindering existing fuzzing tools, finding fewer bugs within a

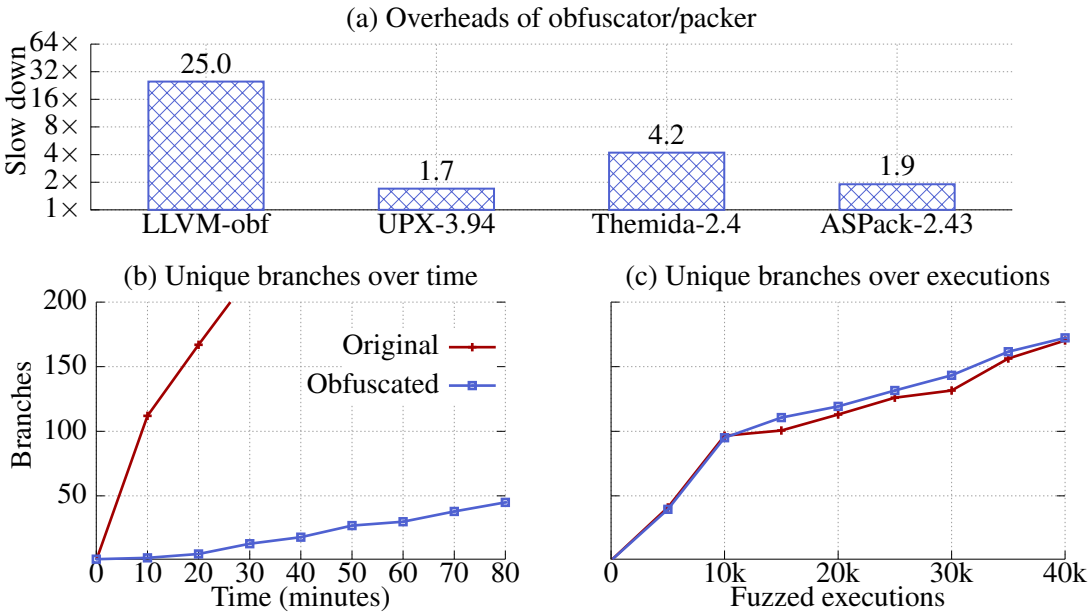

Figure 3.1: Impact of obfuscation techniques on fuzzing. (a) Obfuscation techniques introduce 1.7×-25.0× execution slow down. (b) and (c) fuzzing obfuscated binaries discovers fewer program paths over time, but gets a similar number of paths over executions.

fixed time; second, the protected program should still run efficiently in normal usage; third, the protection code should not be easily identified or removed from the protected binary by straightforward analysis techniques.

No existing technique can achieve all three goals simultaneously. First, software obfuscation techniques, which impede static program analysis by randomizing binary representations, seem to be effective in thwarting fuzzing attempts [11, 12]. However, we find that it falls short of FUZZIFICATION in two ways. Obfuscation introduces unacceptable overhead to normal program executions. Figure 3.1(a) shows that obfuscation slows the execution by at least 1.7 times when using UPX [62] and up to 25.0 times when using LLVM-obfuscator [63]. Also, obfuscation cannot effectively hinder fuzzers in terms of path exploration. It can slow each fuzzed execution, as shown in Figure 3.1(b), but the path discovery per execution is almost identical to that of fuzzing the original binary, as shown in Figure 3.1(c). Therefore, obfuscation is not an ideal FUZZIFICATION technique. Second, software diversification changes the structure and interfaces of the target application to distribute diversified versions [64, 65, 66, 67]. For example, the technique of N-version

software [65] is able to mitigate exploits because attackers often depend on clear knowledge of the program states. However, software diversification is powerless on hiding the original vulnerability from the attacker's analysis; thus it is not a good approach for FUZZIFICATION.

In this thesis, we propose three FUZZIFICATION techniques for developers to protect their programs from malicious fuzzing attempts: SpeedBump, BranchTrap, and AntiHybrid. The SpeedBump technique aims to slow program execution during fuzzing. It injects delays to *cold* paths, which normal executions rarely reach but that fuzzed executions frequently visit. The BranchTrap technique inserts a large number of input-sensitive jumps into the program so that any input drift will significantly change the execution path. This will induce coverage-based fuzzing tools to spend their efforts on injected bug-free paths instead of on the real ones. The AntiHybrid technique aims to thwart hybrid fuzzing approaches that incorporate traditional fuzzing methods with dynamic taint analysis and symbolic execution.

We develop defensive mechanisms to hinder attackers identifying or removing our techniques from protected binaries. For SpeedBump, instead of calling the `sleep` function, we inject randomly synthesized CPU-intensive operations to cold paths and create control-flow and data-flow dependencies between the injected code and the original code. We reuse existing binary code to realize BranchTrap to prevent an adversary from identifying the injected branches.

To evaluate our FUZZIFICATION techniques, we apply them on the LAVA-M dataset and nine real-world applications, including `libjpeg`, `libpng`, `libtiff`, `pcre2`, `readelf`, `objdump`, `nm`, `objcopy`, and `MuPDF`. These programs are extensively used to evaluate the effectiveness of fuzzing tools [68, 69, 70, 71]. Then, we use four popular fuzzers —AFL, HonggFuzz, VUzzer, and QSYM— to fuzz the original programs and the protected ones for the same amount of time. On average, fuzzers detect 14.2 times more bugs from the original binaries and 3.0 times more bugs from the LAVA-M dataset than those from "fuzzified" ones. At the same time, our FUZZIFICATION techniques decrease the total number of discovered paths by 70.3%, and maintain user-specified overhead budget. This result shows that our

FUZZIFICATION techniques successfully decelerate fuzzing performance on vulnerability discovery. We also perform an analysis to show that data-flow and control-flow analysis techniques cannot easily disarm our techniques.

3.2 Background and Problem

3.2.1 Fuzzing Techniques

The goal of fuzzing is to automatically detect program bugs. For a given program, a fuzzer first creates a large number of inputs, either by random mutation or by format-based generation. Then, it runs the program with these inputs to see whether the execution exposes unexpected behaviors, such as a crash or an incorrect result. Compared to manual analysis or static analysis, fuzzing is able to execute the program orders of magnitude more times and thus can explore more program states to maximize the chance of finding bugs.

Fuzzing with Fast Execution

A straightforward way to improve fuzzing efficiency is to make each execution faster. Current research highlights several fast execution techniques, including (1) customized system and hardware to accelerate fuzzed execution and (2) parallel fuzzing to amortize the absolute execution time in large-scale. Among these techniques, AFL uses the fork server and persistent mode to avoid the heavy process creation and can accelerate fuzzing by a factor of two or more [72, 73]. AFL-PT, kAFL, and HonggFuzz utilize hardware features such as Intel Process Tracing (PT) and Branch Trace Store (BTS) to collect code coverage efficiently to guide fuzzing [74, 75, 5]. Recently, Xu *et al*. designed new operating system primitives, like efficient system calls, to speed up fuzzing on multi-core machines [28].

Fuzzing with Coverage-guidance

Coverage-guided fuzzing collects the code coverage for each fuzzed execution and prioritizes fuzzing the input that has triggered new coverage. This fuzzing strategy is based on two

empirical observations: (1) a higher path coverage indicates a higher chance of exposing bugs; and (2) mutating inputs that ever trigger new paths is likely to trigger another new path. Most popular fuzzers take code coverage as guidance, like AFL, HonggFuzz, and LibFuzzer, but with different methods for coverage representation and coverage collection.

Coverage representation. Most fuzzers take basic blocks or branches to represent the code coverage. For example, HonggFuzz and VUzzer use basic block coverage, while AFL instead considers the branch coverage, which provides more information about the program states. Angora [69] combines branch coverage with the call stack to further improve coverage accuracy. However, the choice of representation is a trade-off between coverage accuracy and performance, as more fine-grained coverage introduces higher overhead to each execution and harms the fuzzing efficiency.

Coverage collection. If the source code is available, fuzzers can instrument the target program during compilation or assembly to record coverage at runtime, like in AFL-LLVM mode and LibFuzzer. Otherwise, fuzzers have to utilize either static or dynamic binary instrumentation to achieve a similar purpose, like in AFL-QEMU mode [76]. Also, several fuzzers leverage hardware features to collect the coverage [74, 75, 5]. Fuzzers usually maintain their own data structure to store coverage information. For example, AFL and HonggFuzz use a fixed-size array and VUzzer utilizes a *Set* data structure in Python to store their coverage. However, the size of the structure is also a trade-off between accuracy and performance: an overly small memory cannot capture every coverage change, while an overly large memory introduces significant overhead. For example, AFL's performance drops 30% if the bitmap size is changed from 64KB to 1MB [68].

Fuzzing with Hybrid Approaches

Hybrid approaches are proposed to help solve the limitations of existing fuzzers. First, fuzzers do not distinguish input bytes with different types (*e.g.*, magic number, length specifier) and thus may waste time mutating less important bytes that cannot affect any control

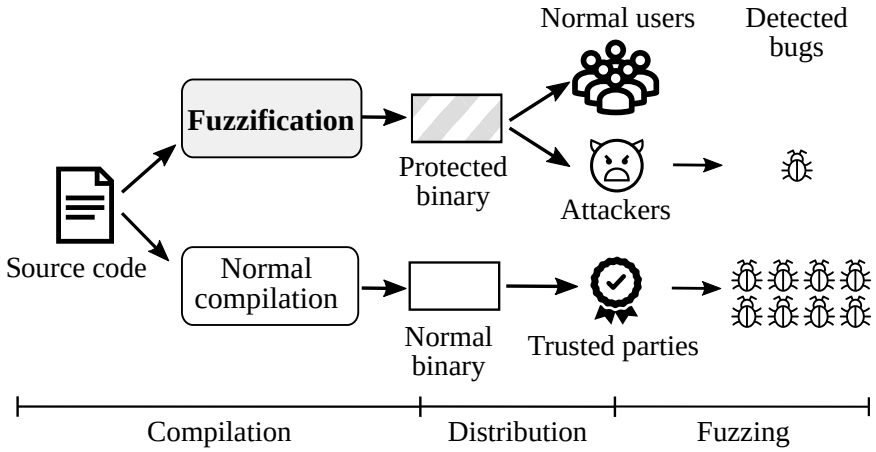

Figure 3.2: Workflow of FUZZIFICATION protection. Developers create a protected binary with FUZZIFICATION techniques and release it to public. Meanwhile, they send the normally compiled binary to trusted parties. Attackers cannot find many bugs from the protected binary through fuzzing, while trusted parties can effectively find significantly more bugs and developers can patch them in time.

flow. In this case, taint analysis is used to help find which input bytes are used to determine branch conditions, like VUzzer [19]. By focusing on the mutation of these bytes, fuzzers can quickly find new execution paths. Second, fuzzers cannot easily resolve complicated conditions, such as comparison with magic value or checksum. Several works [18, 71] utilize symbolic execution to address this problem, which is good at solving complicated constraints but incurs high overhead.

3.2.2 FUZZIFICATION Problem

Program developers may want to completely control the bug-finding process, as any bug leakage can bring attacks and lead to financial loss [77]. They demand exposing bugs by themselves or by trusted parties, but not by malicious end-users. Anti-fuzzing techniques can help to achieve that by decelerating unexpected fuzzing attempts, especially from malicious attackers.

We show the workflow of FUZZIFICATION in Figure 3.2. Developers compile their code in two versions. One is compiled with FUZZIFICATION techniques to generate a protected binary, and the other is compiled normally to generate a normal binary. Then,

Anti-fuzz candidates	Effective	Generic	Efficient	Robust
Pack & obfuscation	✔	✔	✗	✔
Bug injection	✔	✔	✗	✗
Fuzzer identification	✔	✗	✔	✗
Emulator bugs	✔	✗	✔	✔
FUZZIFICATION	✔	✔	✔	✔

Table 3.1: Possible design choices and evaluation with our goals.

developers distribute the protected binary to the public, including normal users and malicious attackers. Attackers fuzz the protected binary to find bugs. However, with the protection of FUZZIFICATION techniques, they cannot find as many bugs quickly. At the same time, developers distribute the normal binary to trusted parties. The trusted parties can launch fuzzing on the normal binary with the native speed and thus can find more bugs in a timely manner. Therefore, developers who receive bug reports from trusted parties can fix the bug before attackers widely abuse it.

Threat Model

We consider motivated attackers who attempt to find software vulnerabilities through state-of-the-art fuzzing techniques, but with limited resources like computing power (at most similar resources as trusted parties). Adversaries have the binary protected by FUZZIFICATION and they have knowledge of our FUZZIFICATION techniques. They can use off-the-shelf binary analysis techniques to disarm FUZZIFICATION from the protected binary. Adversaries who have access to the unprotected binary or even to program source code (*e.g.*, inside attackers, or through code leakage) are out of the scope of this study.

Design Goals and Choices

A FUZZIFICATION technique should achieve the following four goals simultaneously:

- **Effective:** It should effectively reduce the number of bugs found in the protected binary, compared to that found in the original binary.

- **Generic:** It tackles the fundamental principles of fuzzing and is generally applicable to most fuzzers.
- **Efficient:** It introduces minor overhead to the normal program execution.
- **Robust:** It is resistant to the adversarial analysis trying to remove it from the protected binary.

With these goals in mind, we examine four design choices for hindering malicious fuzzing, shown in Table 3.1. Unfortunately, no method can satisfy all goals.

Packing/obfuscation. Software packing and obfuscation are mature techniques against reverse engineering, both generic and robust. However, they usually introduce higher performance overhead to program executions, which not only hinders fuzzing, but also affects the use of normal users.

Bug injection. Injecting arbitrary code snippets that trigger non-exploitable crashes can cause additional bookkeeping overhead and affect end users in unexpected ways [60].

Fuzzer identification. Detecting the fuzzer process and changing the execution behavior accordingly can be bypassed easily (*e.g.*, by changing fuzzer name). Also, we cannot enumerate all fuzzers or fuzzing techniques.

Emulator bugs. Triggering bugs in dynamic instrumentation tools [78, 79, 80] can interrupt fuzzing [81, 82]. However, it requires strong knowledge of the fuzzer, so it is not generic.

3.2.3 Design Overview

We propose three FUZZIFICATION techniques – SpeedBump, BranchTrap, and AntiHybrid– to target each fuzzing technique discussed in §3.2.1. First, SpeedBump injects fine-grained delay primitives into cold paths that fuzzed executions frequently touch but normal executions rarely use (§3.3). Second, BranchTrap fabricates a number of input-sensitive branches to induce the coverage-based fuzzers to waste their efforts on fruitless paths (§3.4). Also, it intentionally saturates the code coverage storage with frequent path collisions so that

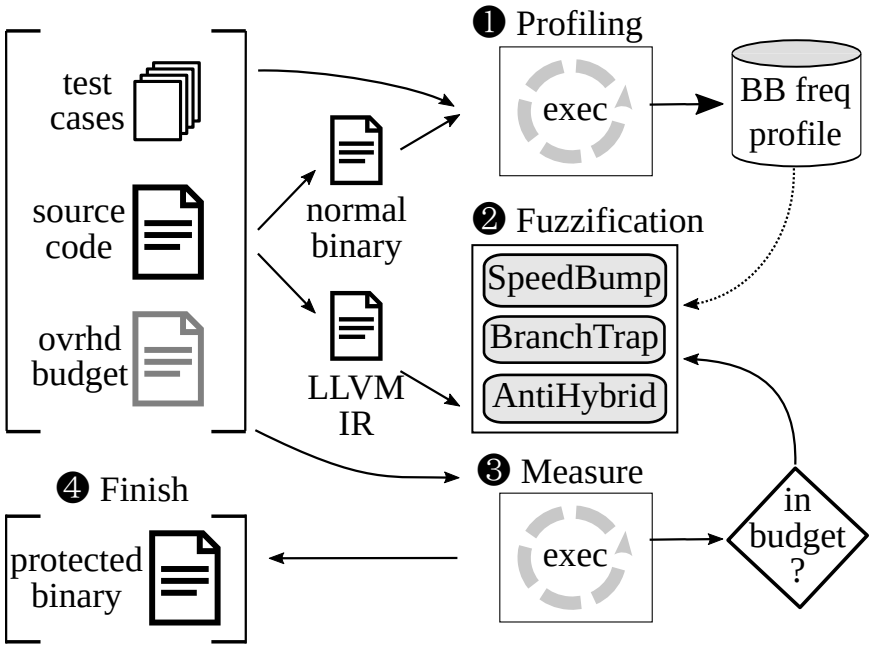

Figure 3.3: Overview of FUZZIFICATION process. It first runs the program with given test cases to get the execution frequency profile. With the profile, it instruments the program with three techniques. The protected binary is released if it satisfies the overhead budget.

the fuzzer cannot identify interesting inputs that trigger new paths. Third, AntiHybrid transforms explicit data-flows into implicit ones to prevent data-flow tracking through taint analysis, and inserts a large number of spurious symbols to trigger path explosion during the symbolic execution (§3.5).

Figure 3.3 shows an overview of our FUZZIFICATION system. It takes the program source code, a set of commonly used test cases, and an overhead budget as input and produces a binary protected by FUZZIFICATION techniques. Note that FUZZIFICATION relies on developers to determine the appropriate overhead budget, whatever they believe will create a balance between the functionality and security of their production. ❶ We compile the program to generate a normal binary and run it with the given normal test cases to collect basic block frequencies. The frequency information tells us which basic blocks are rarely used by normal executions. ❷ Based on the profile, we apply three FUZZIFICATION techniques to the program and generate a temporary protected binary. ❸ We measure the overhead of the temporary binary with the given normal test cases again. If the overhead

is over the budget, we go back to step ❷ to reduce the slow down to the program, such as using shorter delay and adding less instrumentation. If the overhead is far below the budget, we increase the overhead accordingly. Otherwise, ❹ we generate the protected binary.

3.3 SpeedBump: Amplifying Delay in Fuzzing

We propose a technique called SpeedBump to slow the fuzzed execution while minimizing the effect to normal executions. Our observation is that the fuzzed execution frequently falls into paths such as error-handling (*e.g.*, wrong MAGIC bytes) that the normal executions rarely visit. We call them the *cold* paths. Injecting delays in cold paths will significantly slow fuzzed executions but will not affect regular executions that much. We first identify cold paths from normal executions with the given test cases and then inject crafted delays into least-executed code paths. Our tool automatically determines the number of code paths to inject delays and the length of each delay so that the protected binary has overhead under the user-defined budget during normal executions.

Basic block frequency profiling. FUZZIFICATION generates a basic block frequency profile to identify cold paths. The profiling process follows three steps. First, we instrument the target programs to count visited basic blocks during the execution and generate a binary for profiling. Second, with the user-provided test cases, we run this binary and collect the basic blocks visited by each input. Third, FUZZIFICATION analyzes the collected information to identify basic blocks that are rarely executed or never executed by valid inputs. These blocks are treated as cold paths in delay injection.

Our profiling does not require the given test cases to cover 100% of all legitimate paths, but just to trigger the commonly used functionalities. We believe this is a practical assumption, as experienced developers should have a set of test cases covering most of the functionalities (*e.g.*, regression test-suites). Optionally, if developers can provide a set of test cases that trigger uncommon features, our profiling results will be more accurate. For example, for applications parsing well-known file formats (*e.g.*, `readelf` parses ELF

binaries), collecting valid/invalid dataset is straightforward.

Configurable delay injection. We perform the following two steps repeatedly to determine the set of code blocks to inject delays and the length of each delay:

- We start by injecting a 30ms delay to 3% of the least-executed basic blocks in the test executions. We find that this setting is close enough to the final evaluation result.
- We measure the overhead of the generated binary. If it does not exceed the user-defined overhead budget, we go to the previous step to inject more delay into more basic blocks. Otherwise, we use the delay in the previous round as the final result.

Our SpeedBump technique is especially useful for developers who generally have a good understanding of their applications, as well as the requirements for FUZZIFICATION. We provide five options that developers can use to finely tune SpeedBump's effect. Specifically, `MAX_OVERHEAD` defines the overhead budget. Developers can specify any value as long as they feel comfortable with the overhead. `DELAY_LENGTH` specifies the range of delays. We use 10ms to 300ms in the evaluation. `INCLUDE_INCORRECT` determines whether or not to inject delays to error-handling basic blocks (*i.e.*, locations that are *only* executed by invalid inputs), which is enabled by default. `INCLUDE_NON_EXEC` and `NON_EXEC_RATIO` specify whether to inject delays into how ever many basic blocks are never executed during test execution. This is useful when developers do not have a large set of test cases.

Figure 3.4 demonstrates the impact of different options on protecting the `readelf` binary with SpeedBump. We collect 1,948 ELF files on the Debian system as valid test cases and use 600 text and image files as invalid inputs. Figure 3.4(a) shows the maximum ratio of basic blocks that we can inject delay into while introducing overhead less than 1% and 3%. For a 1ms delay, we can instrument 11% of the least-executed basic blocks for a 1% overhead budget and 12% for 3% overhead. For a 120ms delay, we cannot inject any blocks for 1% overhead and can inject only 2% of the cold paths for 3% overhead. Figure 3.4(b) shows the actual performance of AFL-QEMU when it fuzzes SpeedBump-protected binaries. The ratio of injected blocks is determined as in Figure 3.4(a). The result shows that SpeedBump

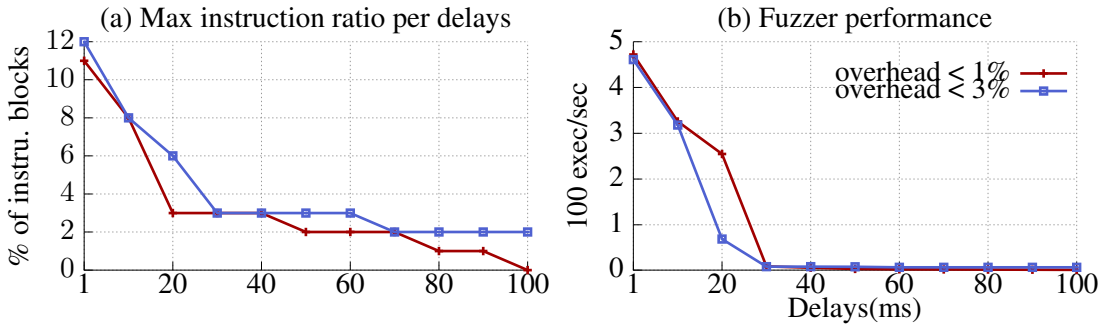

Figure 3.4: Protecting `readelf` with different overhead budgets. While satisfying the overhead budget, (a) demonstrates the maximum ratio of instrumentation for each delay length, and (b) displays the execution speed of AFL-QEMU on protected binaries.

with a 30ms delay slows the fuzzer by more than 50×. Therefore, we use 30ms and the corresponding 3% instrumentation as the starting point.

3.3.1 Analysis-resistant Delay Primitives

As attackers may use program analysis to identify and remove simple delay primitives (*e.g.*, calling `sleep`), we design robust primitives that involve arithmetic operations and are connected with the original code base. Our primitives are based on CSmith [83], which can generate random and bug-free code snippets with refined options. For example, CSmith can generate a function that takes parameters, performs arithmetic operations, and returns a specific type of value. We modified CSmith to generate code that has data dependencies and code dependencies to the original code. Specifically, we pass a variable from the original code to the generated code as an argument, make a reference from the generated code to the original one, and use the return value to modify a global variable of the original code. Figure 3.5 shows an example of our delay primitives. It declares a local variable `PASS_VAR` and modifies global variables `GLOBAL_VAR1` and `GLOBAL_VAR2`. In this way, we introduce data-flow dependency between the original code and the injected code (line 6, 9 and 12), and change the program state without affecting the original program. Although the code is randomly generated, it is tightly coupled with the original code via data-flow and control-flow dependencies. Therefore, it is non-trivial for common binary analysis techniques,

```
//Predefined global variables
int32_t GLOBAL_VAR1 = 1, GLOBAL_VAR2 = 2;
//Randomly generated code
int32_t * func(int32_t p6) {
   int32_t *l0[1000];
   GLOBAL_VAR1 = 0x4507L; // affect global var.
   int32_t *l1 = g8[1][0];
   for (int i = 0; i < 1000; i++)
      l0[i] = p6;         // affect local var from argv.
   (*g7) = func2(g6++);
   (*g5) |= ~(!func3(**g4 = ~0UL));
   return l1;             // affect global var.
}
//Inject above function for delay
int32_t PASS_VAR = 20;
GLOBAL_VAR2 = func(PASS_VAR);
```

Figure 3.5: Example delay primitive. Function `func` updates global variables to build data-flow dependency with original program.

like dead-code elimination, to distinguish it from the original code. We repeatedly run the modified CSmith to find appropriate code snippets that take a specific time (*e.g.*, 10ms) for delay injection.

Safety of delay primitives. We utilize the safety checks from CSmith and FUZZIFICATION to guarantee that the generated code is bug-free. First, we use CSmith's default safety checks, which embed a collection of tests in the code, including integer, type, pointer, effect, array, initialization, and global variable. For example, CSmith conducts pointer analysis to detect any access to an out-of-scope stack variable or null pointer dereference, uses explicit initialization to prevent uninitialized usage, applies math wrapper to prevent unexpected integer overflow, and analyzes qualifiers to avoid any mismatch. Second, FUZZIFICATION also has a separate step to help detect bad side effects (*e.g.*, crashes) in delay primitives. Specifically, we run the code 10 times with fixed arguments and discard it if the execution shows any error. Finally, FUZZIFICATION embeds the generated primitives with the same fixed argument to avoid errors.

Fuzzers aware of error-handling blocks. Recent fuzzing proposals, like VUzzer [19] and T-Fuzz [70], identify error-handling basic blocks through profiling and exclude them from the code coverage calculation to avoid repetitive executions. This may affect the effectiveness

of our SpeedBump technique, which uses a similar profiling step to identify cold paths. Fortunately, the cold paths from SpeedBump include not only error-handling basic blocks, but also rarely executed functional blocks. Further, we use similar methods to identify error-handling blocks from the cold paths and provide developers the option to choose not to instrument these blocks. Thus, our FUZZIFICATION will focus on instrumenting rarely executed functional blocks to maximize its effectiveness.

3.4 BranchTrap: Blocking Coverage Feedback

Code coverage information is widely used by fuzzers to find and prioritize interesting inputs [2, 3, 5]. We can make these fuzzers *diligent fools* if we insert a large number of conditional branches whose conditions are sensitive to slight input changes. When the fuzzing process falls into these branch traps, coverage-based fuzzers will waste their resources to explore (a huge number of) worthless paths. Therefore, we propose the technique of BranchTrap to deceive coverage-based fuzzers by misleading or blocking the coverage feedback.

3.4.1 Fabricating Fake Paths on User Input

The first method of BranchTrap is to *fabricate* a large number of conditional branches and indirect jumps, and inject them into the original program. Each fabricated conditional branch relies on some input bytes to determine to take the branch or not, while indirect jumps calculate their targets based on user input. Thus, the program will take different execution paths even when the input slightly changes. Once a fuzzed execution triggers the fabricated branch, the fuzzer will set a higher priority to mutate that input, resulting in the detection of more fake paths. In this way, the fuzzer will keep wasting its resources (*i.e.*, CPU and memory) to inspect fruitless but bug-free fake paths.

To effectively induce the fuzzers focusing on fake branches, we consider the following four design aspects. First, BranchTrap should fabricate a sufficient number of fake paths to

affect the fuzzing policy. Since the fuzzer generates various variants from one interesting input, fake paths should provide different coverage and be directly affected by the input so that the fuzzer will keep unearthing the trap. Second, the injected new paths introduce minimal overhead to regular executions. Third, the paths in BranchTrap should be deterministic regarding user input, which means that the same input should go through the same path. The reason is that some fuzzers can detect and ignore non-deterministic paths (*e.g.*, AFL ignores one input if two executions with it take different paths). Finally, BranchTrap cannot be easily identified or removed by adversaries.

A trivial implementation of BranchTrap is to inject a jump table and use some input bytes as the index to access the table (*i.e.*, different input values result in different jump targets). However, this approach can be easily nullified by simple adversarial analysis. We design and implement a robust BranchTrap with code-reuse techniques, similar in concept to the well-known return-oriented programming (ROP) [84].

BranchTrap with CFG Distortion

To harden BranchTrap, we diversify the return addresses of each injected branch according to the user input. Our idea is inspired by ROP, which reuses existing code for malicious attacks by chaining various small code snippets. Our approach can heavily distort the program control-flow and makes nullifying BranchTrap more challenging for adversaries. The implementation follows three steps. First, BranchTrap collects function epilogues from the program assembly (generated during program compilation). Second, function epilogues with the same instruction sequence are grouped into one jump table. Third, we rewrite the assembly so that the function will retrieve one of several equivalent epilogues from the corresponding jump table to realize the original function return, using some input bytes as the jump table index. As we replace the function epilogue with a functional equivalent, it guarantees the identical operations as the original program.

Figure 3.6 depicts the internal of the BranchTrap implementation at runtime. For one

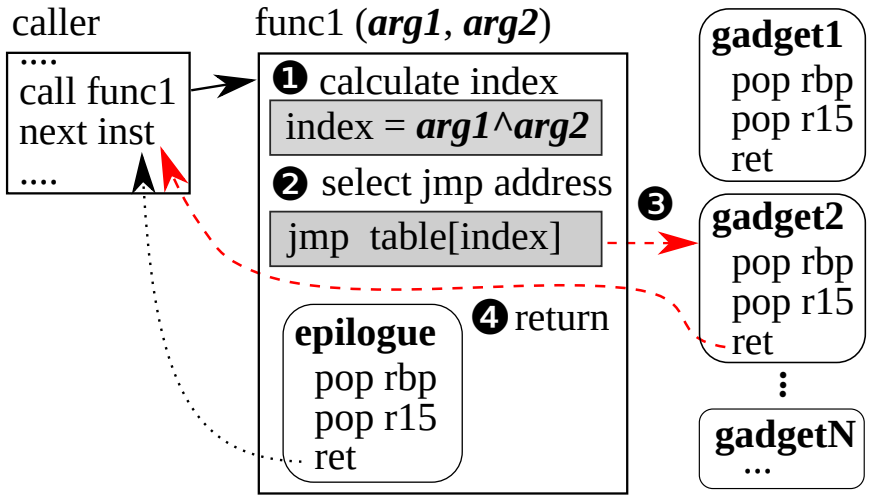

Figure 3.6: BranchTrap by reusing the existing ROP gadgets in the original binary. Among functionally equivalent gadgets, BranchTrap picks the one based on function arguments.

function, BranchTrap ❶ calculates the `XOR`ed value of all arguments. BranchTrap uses this value for indexing the jump table (*i.e.*, candidates for epilogue address). ❷ BranchTrap uses this value as the index to visit the jump table and obtains the concrete address of the epilogue. To avoid out-of-bounds array access, BranchTrap divides the `XOR`ed value by the length of the jump table and takes the remainder as the index. ❸ After determining the target jump address, the control-flow is transferred to the gadget (*e.g.*, the same `pop rbp; pop r15; ret` gadget). ❹ Finally, the execution returns to the original return address.

The ROP-based BranchTrap has three benefitfuzz:

- **Effective:** Control-flow is constantly and sensitively changed together with the user input mutation; thus FUZZIFICATION can introduce a sufficient number of unproductive paths and make coverage feedback less effective. Also, BranchTrap guarantees the same control-flow on the same input (*i.e.*, deterministic path) so that the fuzzer will not ignore these fake paths.
- **Low overhead:** BranchTrap introduces low overhead to normal user operations (*e.g.*, less than 1% overhead) due to its lightweight operations (Store argument; XOR; Resolve jump address; Jump to gadget).
- **Robust:** The ROP-based design significantly increases the complexity for an adversary to identify or patch the binary. We evaluate the robustness of BranchTrap against

adversarial analysis in §3.6.4.

3.4.2 Saturating Fuzzing State

The second method of BranchTrap is to saturate the *fuzzing state*, which blocks the fuzzers from learning the progress in the code coverage. Different from the first method, which induces fuzzers focusing on fruitless inputs, our goal here is to prevent the fuzzers from finding real interesting ones. To achieve this, BranchTrap inserts a massive number of branches to the program, and exploits the coverage representation mechanism of each fuzzer to mask new findings. BranchTrap is able to introduce an extensive number (*e.g.*, 10K to 100K) of deterministic branches to some rarely visited basic blocks. Once the fuzzer reaches these basic blocks, its coverage table will quickly fill up. In this way, most of the newly discovered paths in the following executions will be treated as *visited*, and thus the fuzzer will discard the input that in fact explores interesting paths. For example, AFL maintains a fixed-size bitmap (*i.e.*, 64KB) to track edge coverage. By inserting a large number of distinct branches, we significantly increase the probability of bitmap collision and thus reduce the coverage inaccuracy.

Figure 3.7(a) demonstrates the impact of bitmap saturation on fuzzing `readelf`. Apparently, a more saturated bitmap leads to fewer path discoveries. Starting from an empty bitmap, AFL identifies over 1200 paths after 10 hours of fuzzing. For the 40% saturation rate, it only finds around 950 paths. If the initial bitmap is highly filled, such as 80% saturation, AFL detects only 700 paths with the same fuzzing effort.

Fuzzers with collision mitigation. Recent fuzzers, like CollAFL [68], propose to mitigate the coverage collision issue by assigning a unique identifier to each path coverage (*i.e.*, branch in case of CollAFL). However, we argue that these techniques will not effectively undermine the strength of our BranchTrap technique on saturating coverage storage for two reasons. First, current collision mitigation techniques require program source code to assign unique identifiers during the linking time optimization [68]. In our threat model,

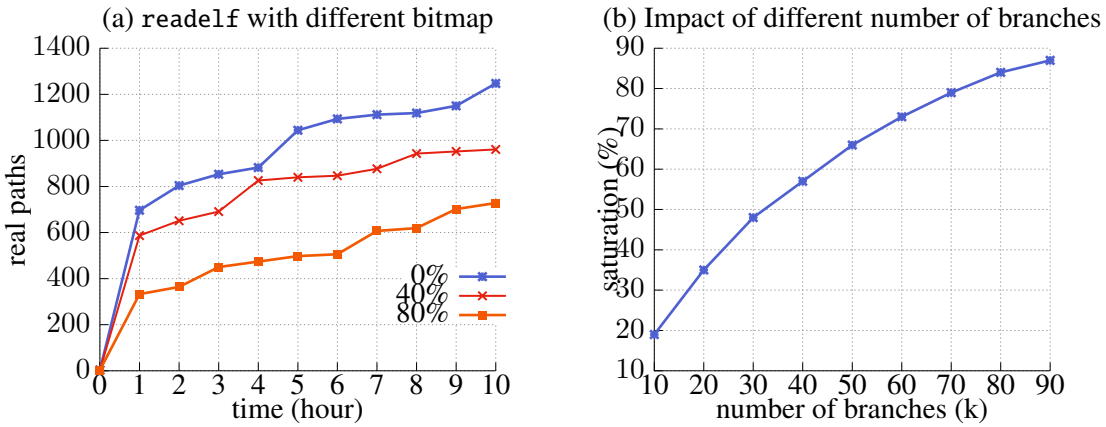

Figure 3.7: Collision during the fuzzing. (a) AFL performance with different initial bitmap saturation. (b) Impact on bitmap with different number of branches.

attackers cannot obtain the program source code or the original binary – they only have a copy of the protected binary, which makes it significantly more challenging to apply similar ID-assignment algorithms. Second, these fuzzers still have to adopt a fixed size storage of coverage because of the overhead of large storage. Therefore, if we can saturate 90% of the storage, CollAFL can only utilize the remaining 10% for ID-assignment; thus the fuzzing performance will be significantly affected.

3.4.3 Design Factors of BranchTrap

We provide developers an interface to configure ROP-based BranchTrap and coverage saturation for optimal protection. First, the number of generated fake paths of ROP-based BranchTrap is configurable. BranchTrap depends on the number of functions to make a distorted control-flow. Therefore, injected BranchTrap is effective when the original program contains plenty of functions. For binaries with fewer functions, we provide an option for developers to split existing basic blocks into multiple ones, each connected with conditional branches. Second, the size of the injected branches for saturating the coverage is also controllable. Figure 3.7(b) shows how the bitmap can be saturated in AFL by increasing the branch number. It clearly shows that more branches can fill up more bitmap entries. For example, 100K branches can fill up more than 90% of a bitmap entry. Injecting a massive

number of branches into the program increases the output binary size. When we inject 100k branches, the size of the protected binary is 4.6MB larger than the original binary. To avoid high code size overhead, we inject a huge number of branches into only one or two of the most rarely executed basic blocks. As long as one fuzzed execution reaches such branches, the coverage storage will be filled and the following fuzzing will find fewer interesting inputs.

3.5 AntiHybrid: Thwarting Hybrid Fuzzers

A hybrid fuzzing method utilizes either symbolic execution or dynamic taint analysis to improve fuzzing efficiency. Symbolic (or concolic) execution is good at solving complicated branch conditions (*e.g.*, magic number and checksum), and therefore can help fuzzers bypass these hard-to-mutate roadblocks. DTA (Dynamic Taint Analysis) helps find input bytes that are related to branch conditions. Recently, several hybrid fuzzing methods have been proposed and successfully discovered security-critical bugs. For example, Driller [18] adapted selective symbolic execution and proved its efficacy during the DARPA Cyber Grand Challenge (CGC). VUzzer [19] utilized dynamic taint analysis to identify path-critical input bytes for effective input mutation. QSYM [71] suggested a fast concolic execution technique that can be scalable on real-world applications.

Nevertheless, hybrid approaches have well-known weaknesses. First, both symbolic execution and taint analysis consume a large amount of resources such as CPU and memory, limiting them to analyzing simple programs. Second, symbolic execution is limited by the path explosion problem. If complex operation is required for processing symbols, the symbolic execution engine has to exhaustively explore and evaluate all execution states; then, most of the symbolic execution engines fail to run to the end of the execution path. Third, DTA analysis has difficulty in tracking implicit data dependencies, such as covert channels, control channels, or timing-based channels. For example, to cover data dependency through a control channel, the DTA engine has to aggressively propagate the taint attribute to any

```
1  char input[] = ...; /* user input */
2  int  value   = ...; /* user input */
3
4  // 1. using implicit data-flow to copy input to antistr
5  //    original code: if (!strcmp(input, "condition!")) { ... }
6  char antistr[strlen(input)];
7  for (int i = 0; i<strlen(input); i++){
8    int ch = 0, temp = 0, temp2 = 0;
9    for (int j = 0; j<8; j++){
10     temp = input[i];
11     temp2 = temp & (1<<j);
12     if (temp2 != 0) ch |= 1<<j;
13   }
14   antistr[i] = ch;
15 }
16 if (!strcmp(antistr, "condition!")) { ... }
17
18 // 2. exploding path constraints
19 //    original code: if (value == 12345)
20 if (CRC_LOOP(value) == OUTPUT_CRC) { ... }
```

Figure 3.8: Example of AntiHybrid techniques. We use implicit data-flow (line 6-15) to copy strings to hinder dynamic taint analysis. We inject hash function around equal comparison (line 20) to cripple symbolic execution engine.

variable after a conditional branch, making the analysis more expensive and the result less accurate.

Introducing implicit data-flow dependencies. We transform the explicit data-flows in the original program into implicit data-flows to hinder taint analysis. FUZZIFICATION first identifies branch conditions and interesting information sinks (*e.g.*, strcmp) and then injects data-flow transformation code according to the variable type. Figure 3.8 shows an example application of AntiHybrid, where array `input` is used to decide branch condition and `strcmp` is an interesting sink function. Therefore, FUZZIFICATION uses implicit data-flows to copy the array (line 6-15) and replaces the original variable to the new one (line 16). Due to the transformed implicit data-flow, the DTA technique cannot identify the correct input bytes that affect the branch condition at line 16.

Implicit data-flow hinders data-flow analysis that tracks direct data propagation. However, it cannot prevent data dependency inference through differential analysis. For example, recent work, RedQueen [85], infers the potential relationship between input and branch con-

Project	Version	Program	Arg.	Seeds	Overhead (Binary size)				Overhead (CPU)			
					Speed	BranchTrap	AntiHybrid	All	Speed	BranchTrap	AntiHybrid	All
libjpeg	2017.7	djpeg		GIT	9.0% (0.1M)	101.5% (1.2M)	0.3% (0.0M)	103.2% (1.3M)	1.5%	0.9%	0.3%	2.4%
libpng	1.6.27	readpng		GIT	6.2% (0.1M)	56.0% (1.3M)	0.9% (0.0M)	65.7% (1.5M)	1.8%	2.0%	0.3%	4.0%
libtiff	4.0.6	tiffinfo		GIT	9.2% (0.2M)	72.5% (1.5M)	0.8% (0.0M)	77.3% (1.6M)	1.0%	2.1%	0.5%	4.8%
pcre2	10	pcre2test		built-in	12.9% (0.2M)	85.3% (1.3M)	0.8% (0.0M)	108.6% (1.7M)	1.2%	1.2%	1.0%	3.1%
binutils	2.23	readelf	-a	ELF files	9.6% (0.2M)	77.3% (1.3M)	0.2% (0.0M)	81.0% (1.4M)	1.0%	0.9%	0.9%	3.1%
		objdump	-d		1.4% (0.1M)	17.0% (1.3M)	0.1% (0.0M)	17.5% (1.3M)	1.6%	2.0%	0.9%	4.6%
		nm			1.9% (0.1M)	23.1% (1.2M)	0.1% (0.0M)	23.3% (1.2M)	1.8%	1.6%	1.1%	4.5%
		objcopy	-S		1.7% (0.1M)	20.2% (1.3M)	0.1% (0.0M)	20.6% (1.3M)	1.7%	0.8%	0.5%	2.9%
Average					6.5%	56.6%	0.4%	62.1%	1.4%	1.4%	0.7%	3.7%

Table 3.2: Code size overhead and performance overhead of fuzzified binaries. `GIT` means Google Image Test-suite. We set performance overhead budget as 5%. For size overhead, we show the percentage and the increased size.

ditions through pattern matching, and thus can bypass the implicit data-flow transformation. However, RedQueen requires the branch condition value to be explicitly shown in the input, which can be easily fooled through simple data modification (*e.g.*, adding the same constant value to both operands of the comparison).

Exploding path constraints. To hinder hybrid fuzzers using symbolic execution, FUZZIFICATION injects multiple code chunks to intentionally trigger path explosions. Specifically, we replace each comparison instruction by comparing the hash values of the original comparison operands. We adopt the hash function because symbolic execution cannot easily determine the original operand with the given hash value. As hash functions usually introduce non-negligible overhead to program execution, we utilize the lightweight cyclic redundancy checking (CRC) loop iteration to transform the branch condition to reduce performance overhead. Although theoretically CRC is not as strong as hash functions for hindering symbolic execution, it also introduces significant slow down. Figure 3.8 shows an example of the path explosion instrumentation. To be specific, FUZZIFICATION changes the original condition `(value == 12345)` to `(CRC_LOOP(value) == OUTPUT_CRC)` (at line 20). If symbolic execution decides to solve the constraint of the CRC, it will mostly return a timeout error due to the complicated mathematics. For example, QSYM, a state-of-the-art fast symbolic execution engine, is armed with many heuristics to scale on real-world applications. When QSYM first tries to solve the complicated constraint that we injected, it will fail due to the timeout or path explosion. Once injected codes are run by the fuzzer multiple times, QSYM identifies the repetitive basic blocks (*i.e.*, injected hash function) and performs *basic block pruning*, which decides not to generate a further constraint from it to assign resources into a new constraint. After that, QSYM will not explore the condition with the injected hash function; thus, the code in the branch can be explored rarely.

Tasks	Target	AFL	HonggFuzz	QSym	VUzzer
Coverage	8 binaries	O,S,B,H,A	O,S,B,H,A	O,S,B,H,A	–
	MuPDF	O,A	O,A	O,A	–
Crash	4 binaries	O,A	O,A	O,A	–
	LAVA-M	O,A	O,A	O,A	O,A

Table 3.3: Experiments summary. Protection optionfuzz: **O**riginal, **S**peedBump, **B**ranchTrap, Anti**H**ybrid, **A**ll. We use 4 binutils binaries, 4 binaries from Google OSS project and `MuPDF` to measure the code coverage. We use binutils binaries and LAVA-M programs to measure the number of unique crashes.

3.6 Evaluation

We evaluate our FUZZIFICATION techniques to understand their effectiveness on hindering fuzzers from exploring program code paths (§3.6.1) and detecting bugs (§3.6.2), their practicality of protecting real-world large programs (§3.6.3), and their robustness against adversarial analysis techniques (§3.6.4).

Implementation. Our FUZZIFICATION framework is implemented in a total of 6,559 lines of Python code and 758 lines of C++ code. We implement the SpeedBump technique as an LLVM pass and use it to inject delays into cold blocks during the compilation. For the BranchTrap, we analyze the assembly code and modify it directly. For the AntiHybrid technique, we use an LLVM pass to introduce the path explosion and utilize a python script to automatically inject implicit data-flows. Currently, our system supports all three FUZZIFICATION techniques on 64bit applications, and is able to protect 32bit applications except for the ROP-based BranchTrap.

Experimental setup. We evaluate FUZZIFICATION against four state-of-the-art fuzzers that work on binaries, specifically, AFL in QEMU mode, HonggFuzz in Intel-PT mode, VUzzer 32[1], and QSYM with AFL-QEMU. We set up the evaluation on two machines, one with Intel Xeon CPU E7-8890 v4@2.20GHz, 192 processors and 504 GB of RAM, and

[1]We also tried to use VUzzer64 to fuzz different programs, but it did not find any crashes even for any original binary after three-day fuzzing. Since VUzzer64 is still experimental, we will try the stable version in the future.

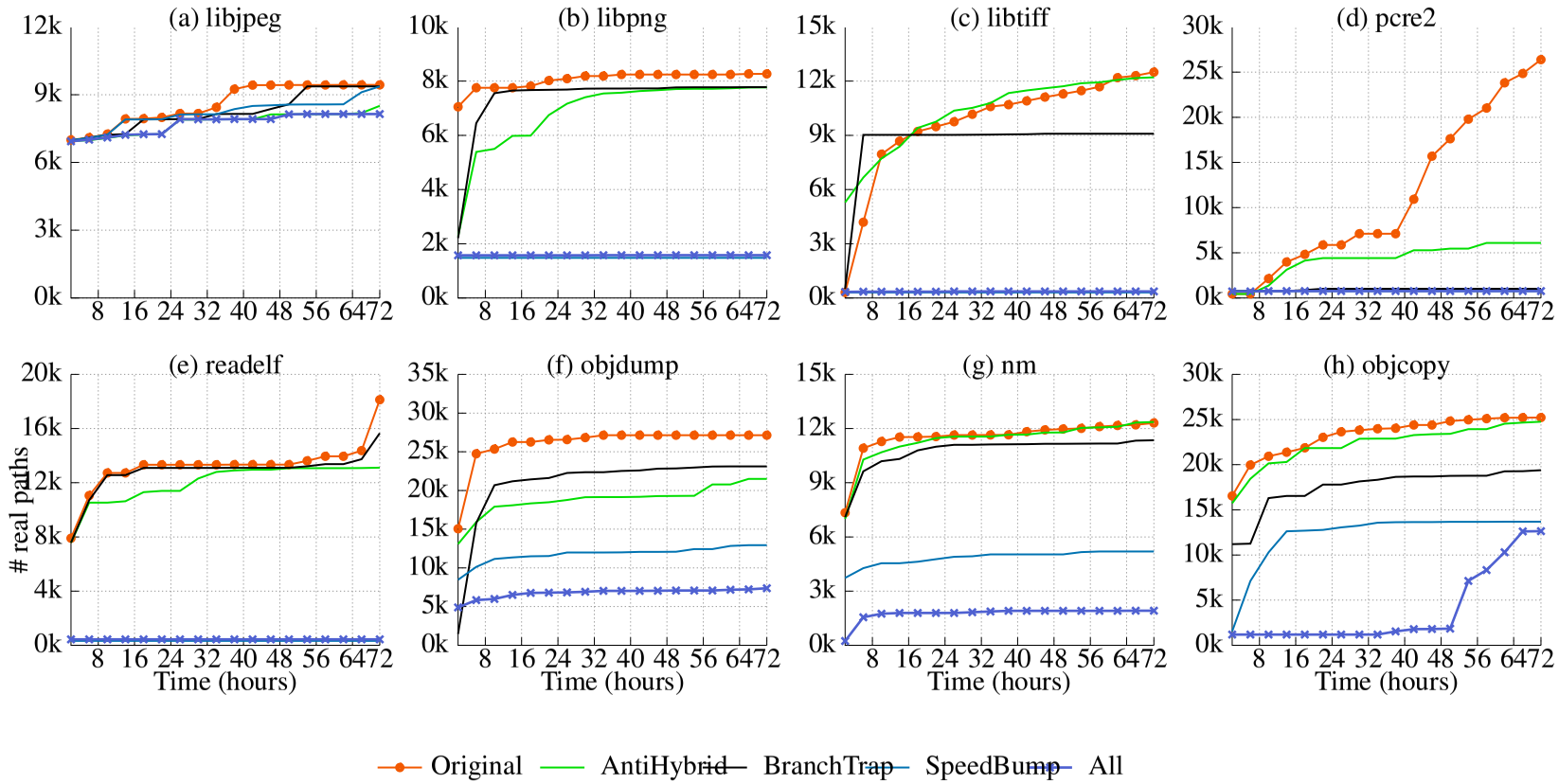

Figure 3.9: Paths discovered by AFL-QEMU from real-world programs. Each program is compiled with five settingfuzz: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and all protections. We fuzz them with AFL-QEMU for three days.

Category	Option	Design Choice
SpeedBump	max_overhead	2%
	delay_length	10ms to 300ms
	include_invalid	True
	include_non_exec	True (5%)
BranchTrap	max_overhead	2%
	bitmap_saturation	40% of 64k bitmap
AntiHybrid	max_overhead	1%
	include_non_exec	True (5%)
Overall	max_overhead	5%

Table 3.4: Our configuration values for the evaluation.

another with Intel Xeon CPU E7-4820@2.00GHz, 32 processors and 128 GB of RAM.

To get reproducible results, we tried to eliminate the non-deterministic factors from fuzzers: we disable the address space layout randomization of the experiment machine and force the deterministic mode for AFL. However, we have to leave the randomness in HonggFuzz and VUzzer, as they do not support deterministic fuzzing. Second, we used the same set of test cases for basic block profiling in FUZZIFICATION, and fed the same seed inputs for different fuzzers. Third, we used identical FUZZIFICATION techniques and configurations when we conducted code instrumentation and binary rewriting for each target application. Last, we pre-generated FUZZIFICATION primitives (*e.g.*, SpeedBump codes for 10ms to 300ms and BranchTrap codes with deterministic branches), and used the primitives for all protections. Note that developers should use different primitives for the actual releasing binary to avoid code pattern matching analysis.

Target applications. We select the LAVA-M data set [86] and nine real-world applications as the fuzzing targets, which are commonly used to evaluate the performance of fuzzers [69, 68, 28, 19]. The nine real-world programs include four applications from the Google fuzzer test-suite [30], four programs from the binutils [87] (shown in Table 3.2), and the PDF reader `MuPDF`. We perform two sets of experiments on these binaries, summarized in

Table 3.3. First, we fuzz nine real-world programs with three fuzzers (all except VUzzer[2]) to measure the impact of FUZZIFICATION on finding code paths. Specifically, we compile eight real-world programs (all except `MuPDF`) with five different settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and a combination of three techniques (full protection). We compile `MuPDF` with two settings for simplicity: no protection and full protection. Second, we use three fuzzers to fuzz four binutils programs and all four fuzzers to fuzz LAVA-M programs to evaluate the impact of FUZZIFICATION on unique bug finding. All fuzzed programs in this step are compiled in two versions: with no protection and with full protection. We compiled the LAVA-M program to a 32bit version in order to be comparable with previous research. Table 3.4 shows the configuration of each technique used in our compilation. We changed the fuzzer's timeout if the binaries cannot start with the default timeout (*e.g.*, 1000 ms for AFL-QEMU).

Evaluation metric. We use two metrics to measure the effectiveness of FUZZIFICATION: code coverage in terms of discovered *real path*s, and unique crashes. Real path is the execution path shown in the original program, excluding the fake ones introduced by BranchTrap. We further excluded the real paths triggered by seed inputs so that we can focus on the ones discovered by fuzzers. Unique crash is measured as the input that can make the program crash with a distinct real path. We filter out duplicate crashes that are defined in AFL [88] and are widely used by other fuzzers [69, 39].

3.6.1 Reducing Code Coverage

Impact on Normal Fuzzers

We measure the impact of FUZZIFICATION on reducing the number of real paths against AFL-QEMU and HonggFuzz-Intel-PT. Figure 3.9 shows the 72-hour fuzzing result from AFL-QEMU on different programs with five protection settings. The result of HonggFuzz-

[2]Due to time limit, we only use VUzzer 32 to finding bugs from LAVA-M programs. We plan to do other evaluations in the future.

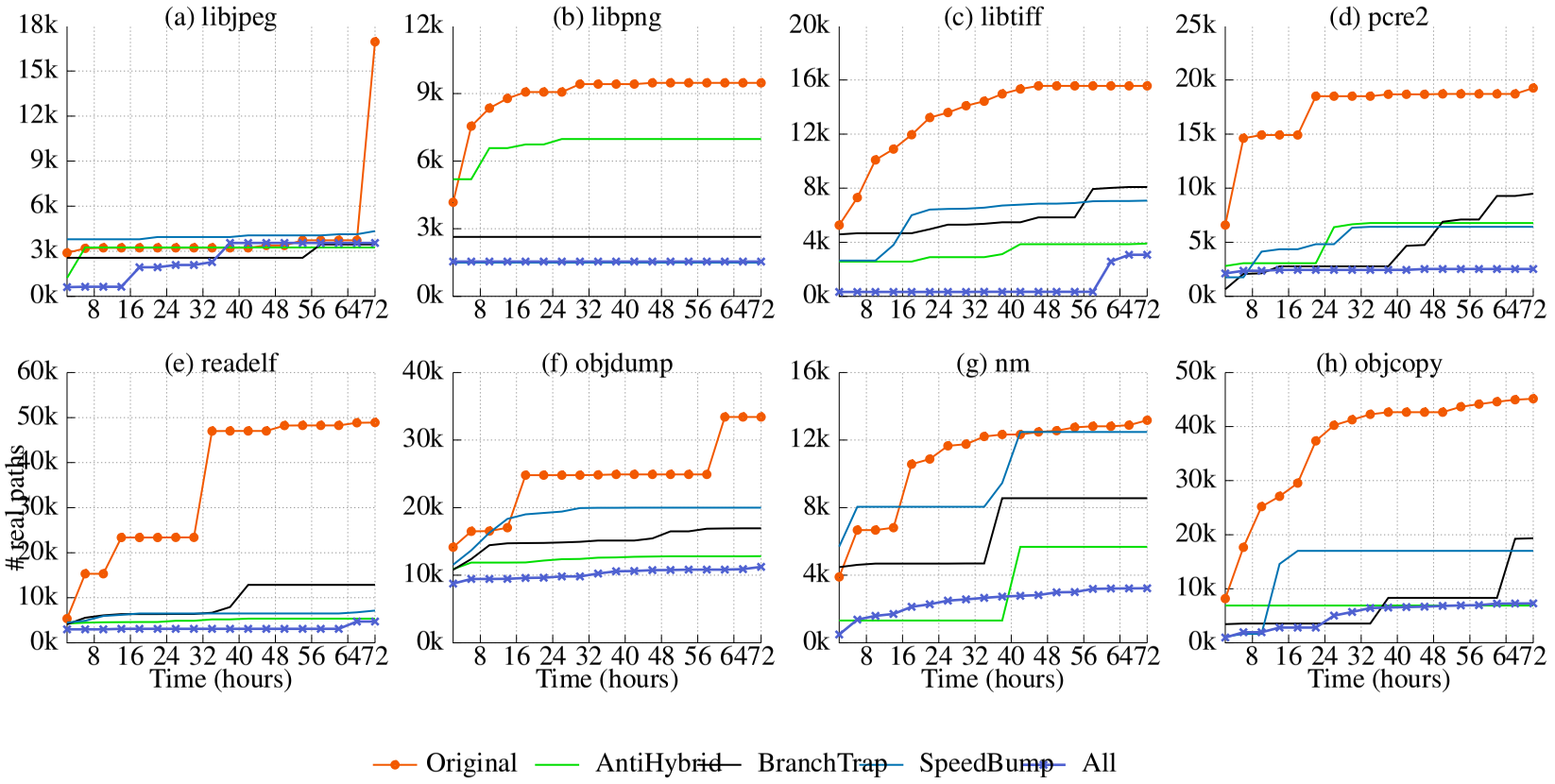

Figure 3.10: Paths discovered by QSYM from real-world programs. Each program is compiled with the same five settings as in Figure 3.9. We fuzz these programs for three days, using QSYM as the symbolic execution engine and AFL-QEMU as the native fuzzer.

Intel-PT is similar.

In summary, with all three techniques, FUZZIFICATION can reduce discovered real paths by 76% to AFL, and by 67% to HonggFuzz, on average. For AFL, the reduction rate varies from 14% to 97% and FUZZIFICATION reduces over 90% of path discovery for `libtiff`, `pcre2` and `readelf`. For HonggFuzz, the reduction rate is between 38% to 90% and FUZZIFICATION only reduces more than 90% of paths for `pcre2`. As FUZZIFICATION automatically determines the details for each protection to satisfy the overhead budget, its effect varies for different programs.

Table 3.5 shows the effect of each technique on hindering path discovery. Among them, SpeedBump achieves the best protection against normal fuzzers, followed by BranchTrap and AntiHybrid. Interestingly, although AntiHybrid is developed to hinder hybrid approaches, it also helps reduce the discovered paths in normal fuzzers. We believe this is mainly caused by the slow down in fuzzed executions.

We measured the overhead by different FUZZIFICATION techniques, on program size and execution speed. The result is given in Table 3.2. In summary, FUZZIFICATION satisfies the user-specified overhead budget, but shows relatively high space overhead. On average, binaries armed with FUZZIFICATION are 62.1% larger than the original ones. The extra code mainly comes from the BranchTrap technique, which inserts massive branches to achieve bitmap saturation. Note that the extra code size is almost the same across different programs. Therefore, the size overhead is high for small programs, but is negligible for large applications. For example, the size overhead is less than 1% for LibreOffice applications, as we show in Table 3.7. Further, BranchTrap is configurable, and developers may inject a smaller number of fake branches to small programs to avoid large-size overhead.

Analysis on less effective results. FUZZIFICATION shows less effectiveness on protecting the `libjpeg` application. Specifically, it decreases the number of real paths on `libjpeg` by 13% to AFL and by 37% to HonggFuzz, whereas the average reduction is 76% and 67%, respectively. We analyzed FUZZIFICATION on `libjpeg` and find that SpeedBump and

	SpeedBump	BranchTrap	AntiHybrid	All
AFL-QEMU	-66%	-23%	-18%	-74%
HonggFuzz (PT)	-44%	-14%	-7%	-61%
QSym (AFL-QEMU)	-59%	-58%	-67%	-80%
Average	-56%	-31%	-30%	-71%

Table 3.5: Reduction of discovered paths by FUZZIFICATION techniques. Each value is an average of the fuzzing result from eight real-world programs, as shown in Figure 3.9 and Figure 3.10.

BranchTrap cannot effectively protect `libjpeg`. Specifically, these two techniques only inject nine basic blocks within the user-specified overhead budget (2% for SpeedBump and 2% for BranchTrap), which is less than 0.1% of all basic blocks. To address this problem, developers may increase the overhead budget so that FUZZIFICATION can insert more roadblocks to protect the program.

Impact on Hybrid Fuzzers

We also evaluated FUZZIFICATION's impact on code coverage against QSYM, a hybrid fuzzer that utilizes symbolic execution to help fuzzing. Figure 3.10 shows the number of real paths discovered by QSYM from the original and protected binaries. Overall, with all three techniques, FUZZIFICATION can reduce the path coverage by 80% to QSYM on average, and shows consistent high effectiveness on all tested programs. Specifically, the reduction rate varies between 66% (`objdump`) to 90% (`readelf`). The result of libjpeg shows an interesting pattern: QSYM finds a large number of real paths from the original binary in the last 8 hours, but it did not get the same result from any protected binary. Table 3.5 shows that AntiHybrid achieves the best effect (67% path reduction) against hybrid fuzzers, followed by SpeedBump (59%) and BranchTrap (58%).

Comparison with normal fuzzing result. QSYM uses efficient symbolic execution to help find new paths in fuzzing, and therefore it is able to discover 44% more real paths than AFL from original binaries. As we expect, AntiHybrid shows the most impact on QSYM (67% reduction), and less effect on AFL (18%) and HonggFuzz (7%). With our FUZZIFICATION

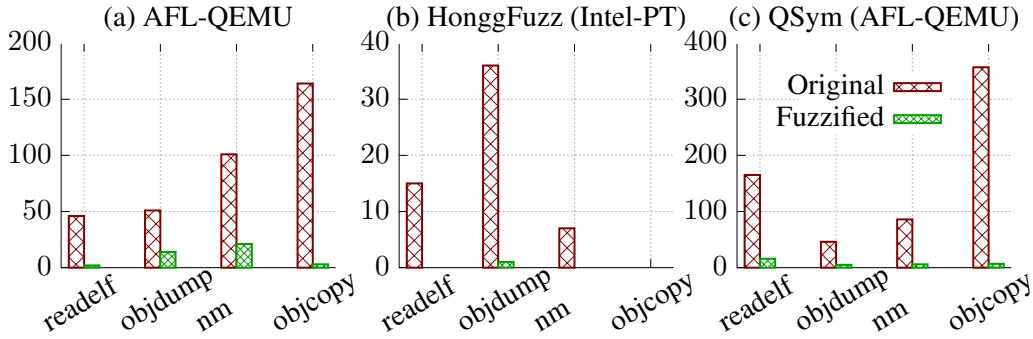

Figure 3.11: Crashes found by different fuzzers from binutils programs. Each program is compiled as original (no protection) and fuzzified (three techniques) and is fuzzed for three days.

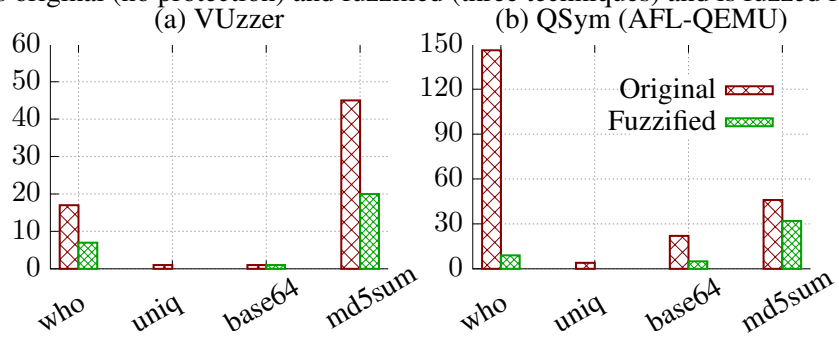

Figure 3.12: Bugs found by VUzzer and QSYM from LAVA-M dataset. HonggFuzz discovers three bugs from the original `uniq`. AFL does not find any bug.

techniques, QSYM shows less advantage over normal fuzzers, reduced from 44% to 12%.

3.6.2 Hindering Bug Finding

We measure the number of unique crashes that fuzzers find from the original and protected binaries. Our evaluation first fuzzes four binutils programs and LAVA-M applications with three fuzzers (all but VUzzer). Then we fuzz LAVA-M programs with VUzzer, where we compiled them into 32bit versions and excluded the protection of ROP-based BranchTrap, which is not implemented yet for 32bit programs.

Impact on Real-World Applications

Figure 3.11 shows the total number of unique crashes discovered by three fuzzers in 72 hours. Overall, FUZZIFICATION reduces the number of discovered crashes by 93%, specifically, by 88% to AFL, by 98% to HonggFuzz, and by 94% to QSYM. If we assume a consistent crash-discovery rate along the fuzzing process, fuzzers have to take 40 times more effort

to detect the same number of crashes from the protected binaries. As the crash-discovery rate usually reduces over time in real-world fuzzing, fuzzers will have to take much more effort. Therefore, FUZZIFICATION can effectively hinder fuzzers and makes them spend significantly more time discovering the same number of crash-inducing inputs.

Impact on LAVA-M Dataset

Compared with other tested binaries, LAVA-M programs are smaller in size and simpler in operation. If we inject a 1ms delay on 1% of rarely executed basic block on `who` binary, the program will suffer a slow down of more than 40 times. To apply FUZZIFICATION on the LAVA-M dataset, we allow higher overhead budget and apply more fine-grained FUZZIFICATION. Specifically, we used tiny delay primitives (*i.e.*, 10 µs to 100 µs), tuned the ratio of basic block instrumentation from 1% to 0.1%, reduced the number of applied AntiHybrid components, and injected smaller deterministic branches to reduce the code size overhead. Table 3.6 shows the run-time and space overhead of the generated LAVA-M programs with FUZZIFICATION techniques.

After fuzzing the protected binaries for 10 hours, AFL-QEMU does not find any crash. HonggFuzz detects three crashes from the original `uniq` binary and cannot find any crash from any protected binary. Figure 3.12 illustrates the fuzzing result of VUzzer and QSYM. Overall, FUZZIFICATION can reduce 56% of discovered bugs to VUzzer and 78% of discovered bugs to QSYM. Note that the fuzzing result on the original binaries is different from the ones reported in the original papers [71, 19] for several reason: VUzzer and QSYM cannot eliminate non-deterministic steps during fuzzing; we run the AFL part of each tool in QEMU mode; LAVA-M dataset is updated with several bug fixes[3].

	who	uniq	base64	md5sum	Average
Overhead (Size)	17.1% (0.3M)	220.6% (0.3M)	220.0% (0.3M)	210.7% (0.3M)	167.1%
Overhead (CPU)	22.7%	13.2%	21.1%	6.5%	15.9%

Table 3.6: Overhead of FUZZIFICATION on LAVA-M binaries. The overhead is higher as LAVA-M binaries are relatively small (*e.g.*, $\approx$ 200KB).

Category	Program	Version	Overhead	
			Size	CPU
LibreOffice	Writer	6.2	< 1% (+1.3 MB)	0.4%
	Calc		< 1% (+1.3 MB)	0.4%
	Impress		< 1% (+1.3 MB)	0.2%
Music Player	Clementine	1.3	4.3% (+1.3 MB)	0.5%
PDF Reader	MuPDF	1.13	4.1% (+1.3 MB)	2.2%
Image Viewer	Nomacs	3.10	21% (+1.2 MB)	0.7%
Average			5.4%	0.73%

Table 3.7: FUZZIFICATION on GUI applications. The CPU overhead is calculated on the application launching time. Due to the fixed code injection, code size overhead is negligible for these large applications.

3.6.3 Anti-fuzzing on Realistic Applications

To understand the practicality of FUZZIFICATION on large and realistic applications, we choose six programs that have a graphical user interface (GUI) and depend on tens of libraries. As fuzzing large and GUI programs is a well-known challenging problem, our evaluation here focuses on measuring the overhead of FUZZIFICATION techniques and the functionality of protected programs. When applying the SpeedBump technique, we have to skip the basic block profiling step due to the lack of command-line interface (CLI) support (*e.g.*, `readelf` parses ELF file and displays results in command line); thus, we only insert slow down primitives into error-handling routines. For the BranchTrap technique, we choose to inject massive fake branches into basic blocks near the entry point. In this way, the program execution will always pass the injected component so that we can measure

[3] httpfuzz://github.com/panda-re/lava/search?q=bugfix&type=Commits

	Pattern matching	Control analysis	Data analysis	Manual analysis
SpeedBump	✔	✔	✔	-
BranchTrap	✔	✔	✔	-
AntiHybrid	-	✔	✔	-

Table 3.8: Defense against adversarial analysis. ✔ indicates that the FUZZIFICATION technique is resistant to that adversarial analysis.

runtime overhead correctly. We apply the AntiHybrid technique directly.

For each protected application, we first manually run it with multiple inputs, including given test cases, and confirm that FUZZIFICATION does not affect the program's original functionality. For example, `MuPDF` successfully displays, edits, saves, and prints all tested PDF documents. Second, we measure the code size and runtime overhead of the protected binaries for given test cases. As shown in Table 3.7, on average, FUZZIFICATION introduces 5.4% code size overhead and 0.73% runtime overhead. Note that the code size overhead is much smaller than that of previous programs (*i.e.*, 62.1% for eight relatively small programs Table 3.2 and over 100% size overhead for simple LAVA-M programs Table 3.6).

Anti-fuzzing on MuPDF. We also evaluated the effectiveness of FUZZIFICATION on protecting `MuPDF` against three fuzzers – AFL, HonggFuzz, and QSYM– as `MuPDF` supports the CLI interface through the tool called "mutool." We compiled the binary with the same parameter shown in Table 3.4 and performed basic block profiling using the CLI interface. After 72-hours of fuzzing, no fuzzer finds any bug from `MuPDF`. Therefore, we instead compare the number of real paths between the original binary and the protected one. As shown in Figure 3.13, FUZZIFICATION reduces the total paths by 55% on average, specifically, by 77% to AFL, by 36% to HonggFuzz, and 52% to QSYM. Therefore, we believe it is more challenging for real-world fuzzers to find bugs from protected applications.

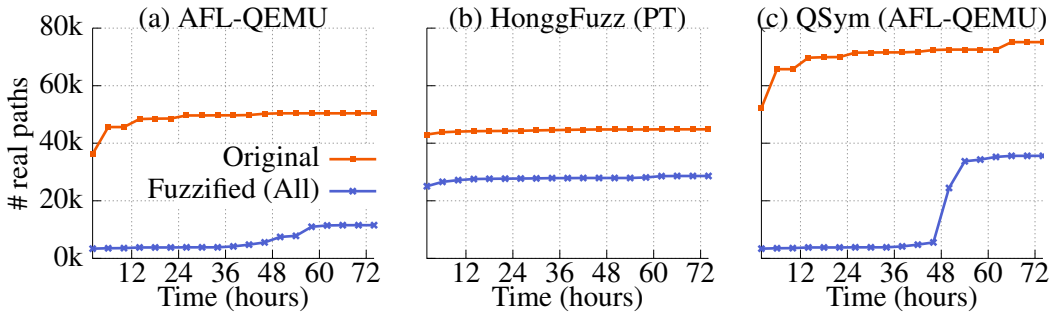

Figure 3.13: Testing `MuPDF`. Paths discovered by different fuzzers from the original `MuPDF` and the one protected by three FUZZIFICATION techniques.

3.6.4 Evaluating Best-effort Countermeasures

We evaluate the robustness of FUZZIFICATION techniques against off-the-shelf program analysis techniques that adversaries may use to reverse our protections. However, the experiment results do not particularly indicate that FUZZIFICATION is robust against strong adversaries with incomparable computational resources.

Table 3.8 shows the analysis we covered and summarizes the evaluation result. First, attackers may search particular code patterns from the protected binary in order to identify injected protection code. To test anti-fuzzing against pattern matching, we examine a number of code snippets that are repeatedly used throughout the protected binaries. We found that the injected code by AntiHybrid crafts several observable patterns, like hash algorithms or data-flow reconstruction code, and thus could be detected by attackers. One possible solution to this problem is to use existing diversity techniques to eliminate the common patterns [64]. We confirm that no specific patterns can be found in SpeedBump and BranchTrap because we leverage CSmith [83] to randomly generate a new code snippet for each FUZZIFICATION process.

Second, control-flow analysis can identify unused code in a given binary automatically and thus automatically remove it (*i.e.*, dead code elimination). However, this technique cannot remove our FUZZIFICATION techniques, as all injected code is cross-referenced with the original code. Third, data-flow analysis is able to identify the data dependency. We run protected binaries inside the debugging tool, GDB, to inspect data dependencies between

the injected code and the original code. We confirm that data dependencies always exist via global variables, arguments, and the return values of injected functions. Finally, we consider an adversary who is capable of conducting manual analysis for identifying the anti-fuzzing code with the knowledge of our techniques. It is worth noting that we do not consider strong adversaries who are capable of analyzing the application logic for vulnerability discovery. Since FUZZIFICATION injected codes are supplemental to the original functions, we conclude that the manual analysis can eventually identify and nullify our techniques by evaluating the actual functionality of the code. However, since the injected code is functionally similar to normal arithmetic operations and has control- and data-dependencies on the original code, we believe that the manual analysis is time-consuming and error-prone, and thus we can deter the time for revealing real bugs.

3.7 Discussion and Future Work

In this section, we discuss the limitations of FUZZIFICATION and suggest provisional countermeasures against them.

Complementing attack mitigation system. The goal of anti-fuzzing is not to completely hide all vulnerabilities from adversaries. Instead, it introduces an expensive cost on the attackers' side when they try to fuzz the program to find bugs, and thus developers are able to detect bugs first and fix them in a timely manner. Therefore, we believe our anti-fuzzing technique is an important complement to the current attack mitigation ecosystem. Existing mitigation efforts either aim to avoid program bugs (*e.g.*, through type-safe language [89, 90]) or aim to prevent successful exploits, assuming attackers will find bugs anyway (*e.g.*, through control-flow integrity [91, 92, 93]). As none of these defenses can achieve 100% protection, our FUZZIFICATION techniques provide another level of defense that further enhances program security. However, we emphasize that FUZZIFICATION alone cannot provide the best security. Instead, we should keep working on all aspects of system security toward a completely secure computer system, including but not limited to secure development

process, effective bug finding, and efficient runtime defense.

Best-effort protection against adversarial analysis. Although we examined existing generic analyses and believe they cannot completely disarm our FUZZIFICATION techniques, the defensive methods only provide a best-effort protection. First, if attackers have almost unlimited resources, such as when they launch APT (advanced persistent threat) attacks, no defense mechanism can survive the powerful adversarial analysis. For example, with extremely powerful binary-level control-flow analysis and data-flow analysis, attackers may finally identify the injected branches by BranchTrap and thus reverse it for an unprotected binary. However, it is hard to measure the amount of required resources to achieve this goal, and meanwhile, developers can choose more complicated branch logic to mitigate reversing. Second, we only examined currently existing techniques and cannot cover all possible analyses. It is possible that attackers who know the details of our FUZZIFICATION techniques propose a specific method to effectively bypass the protection, such as by utilizing our implementation bugs. But in this case, the anti-fuzzing technique will also get updated quickly to block the specific attack once we know the reversing technique. Therefore, we believe the anti-fuzzing technique will get improved continuously along the back-and-forth attack and defense progress.

Trade-off performance for security. FUZZIFICATION improves software security at the cost of a slight overhead, including code size increase and execution slow down. A similar trade-off has been shown in many defense mechanisms and affects the deployment of defense mechanisms. For example, address space layout randomization (ASLR) has been widely adopted by modern operating systems due to small overhead, while memory safety solutions still have a long way to go to become practical. Fortunately, the protection by FUZZIFICATION is quite flexible, where we provide various configuration options for developers to decide the optimal trade-off between security and performance, and our tool will automatically determine the maximum protection under the overhead budget.

3.8 Conclusion

We propose a new attack mitigation system, called FUZZIFICATION, for developers to prevent adversarial fuzzing. We develop three principled ways to hinder fuzzing: injecting delays to slow fuzzed executions; inserting fabricated branches to confuse coverage feedback; transforming data-flows to prevent taint analysis and utilizing complicated constraints to cripple symbolic execution. We design robust anti-fuzzing primitives to hinder attackers from bypassing FUZZIFICATION. Our evaluation shows that FUZZIFICATION can reduce paths exploration by 70.3% and reduce bug discovery by 93.0% for real-world binaries, and reduce bug discovery by 67.5% for LAVA-M dataset.

CHAPTER 4

WINNIE: FUZZING WINDOWS APPLICATIONS WITH HARNESS SYNTHESIS AND FAST CLONING

4.1 Introduction

Fuzzing is an emerging software-testing technique for automatically validating program functionalities and uncovering security vulnerabilities [1]. It randomly mutates program inputs to generate a large corpus and feeds each input to the program. It monitors the execution for abnormal behaviors, like crashing, hanging, or failing security checks [94]. Recent fuzzing efforts have found thousands of vulnerabilities in open-source projects [23, 24, 25, 26]. There are continuous efforts to make fuzzing faster [95, 19, 96] and smarter [28, 71, 18].

However, existing fuzzing techniques are mainly applied to Unix-like OSes, and few of them work as well on Windows platforms. Unfortunately, Windows applications are not free from bugs. Recent report shows that in the past 12 years, 70% of all security vulnerabilities on Windows systems are memory safety issues [13]. In fact, due to the dominance of Windows operating system, its applications remain the most lucrative targets for malicious attackers [14, 15, 16, 17]. To bring popular fuzzing techniques to the Windows platform, we investigate common applications and state-of-the-art fuzzers, and identify three challenges of fuzzing applications on Windows: a predominance of graphical applications, a closed-source ecosystem (e.g., third-party or legacy libraries), and the lack of fast cloning machinery like `fork` on Unix-like OSes.

Windows applications heavily rely on GUIs (graphical user interfaces) to interact with end-users, which poses a major obstacle to fuzzing. As shown in Figure 4.1, XnView [97] requires the user to provide a file through the graphical dialog window. When the user specifies the file path, the main executable parses the file, determines which library to delegate to, and dynamically loads the necessary library to handle the input. Although some efforts try to automate the user interaction [98], the execution speed is much slower than ordinary fuzzing. For example, fuzzing GUI applications with AutoIt yields only around three executions per second [99], whereas Linux fuzzing often achieves speeds of more

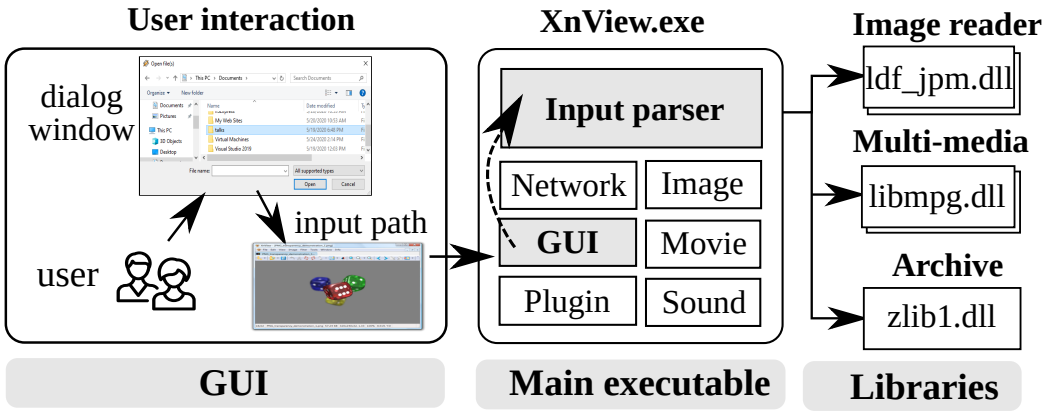

Figure 4.1: Architecture of XnView on Windows. The program accepts the user input via the GUI. The main executable parses the received path and dynamically loads the library to process the input. A fuzzing harness bypasses the GUI to reach the functionality we wish to test.

than 1,000 executions per second. Speed is crucial for effective fuzzing, and this slow-down renders fuzzing GUI application impractical.

The general way to overcome the troublesome GUI is to write a simple program, called a *harness*, to assist with fuzzing. A harness replaces the GUI with a CLI (command-line interface), prepares the execution context such as arguments and memory, and invokes the target functions directly. In this way, we can test the target program without any user interaction. For example, with a harness that receives the input path from the command line and loads the `decoder` library, we can test XnView without worrying about the dialog window. Recent work has even explored generating harnesses automatically for open-source programs [100, 101].

Nevertheless, Windows fuzzing still relies largely on human effort to create effective harnesses because most Windows programs are closed-source, commercial-off-the-shelf (COTS) software [102, 103, 104, 105, 106]. Existing automatic harness synthesis methods require to access the source code, and thus cannot handle closed-source programs easily [100, 101]. Without the source code, we have little knowledge of the program's internals, like the locations of interesting functions and their prototypes. Since manual analysis is error-prone and unscalable to a large number of programs, we need a new method to generate fuzzing harnesses directly from the *binary*.

Finally, Windows lacks the fast cloning machinery (e.g., `fork` syscall) that greatly aids fuzzing on Unix-like OSes. Linux fuzzers like AFL place a fork-server before the target function, and subsequent executions reuse the pre-initialized state by forking. The fork-server makes AFL run 1.5×–2× faster on Linux [107]. `fork` also improves the stability of testing as each child process runs in its own address space, containing any side-effects, like crashes or hangs. However, the Windows kernel does not expose a clear counterpart for `fork`, nor any suitable alternatives. As a result, fuzzers have to re-execute the program from the beginning for each new input, leading to a low execution speed. Although we can write a harness to test the program in a big loop (aka., *persistent mode* [73]), testing many inputs in one process harms stability. For example, each execution may gradually pollute the global program state, eventually leading to divergence and incorrect behavior.

We propose an end-to-end system, WINNIE, to address the aforementioned challenges and make fuzzing Windows programs more practical. WINNIE contains two components: a harness generator that automatically synthesizes harnesses from the program binary alone, and an efficient Windows fork-server. To construct plausible harnesses, our harness generator combines both dynamic and static analysis. We run the target program against several inputs, collect execution traces, and identify interesting functions and libraries that are suitable for fuzzing. Then, our generator searches the execution traces to collect all function calls to candidate libraries, and extracts them to form a harness skeleton. Finally, we try to identify the relationships between different function calls and arguments to build a full harness. Meanwhile, to implement an efficient fork-server for Windows systems, we identified and analyzed undocumented Windows APIs that effectively support a Copy-on-Write `fork` operation similar to the corresponding system call on Unix systems. We established the requirements to use these APIs in a stable manner. The availability of `fork` eliminates the need for existing, crude fuzzing techniques like persistent mode. To the best of our knowledge, this is the first practical counterpart of `fork` on Windows systems for fuzzing.

We implemented WINNIE in 7.1K lines of code (LoC). We applied WINNIE on 59 exe-

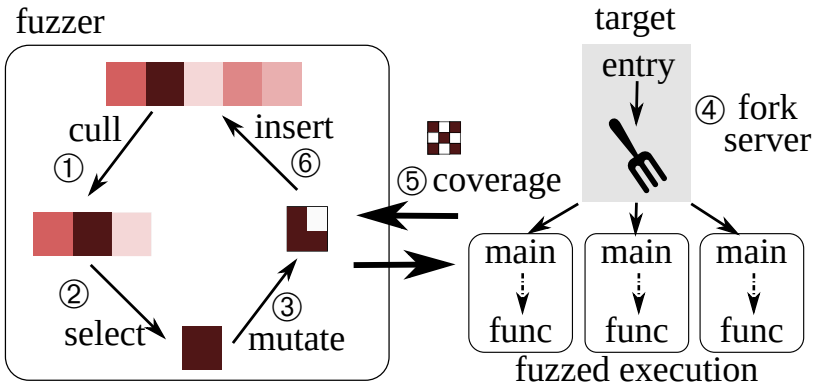

Figure 4.2: Fuzzing overview. (1) The fuzzer maintains a queue of inputs. Each cycle, (2) it picks one input from the queue and (3) modifies it to generate a new input. (4) It feeds the new input into the fuzzed program and (5) records the code coverage. (6) If the execution triggers more coverage, the new input is added back into the queue.

cutables, including Visual Studio, ACDSee, ultraISO and EndNote. Our harness generator automatically synthesized candidate harnesses from execution traces, and 95% of them could be fuzzed directly with only minor modifications (*i.e.*, $\leq$ 10 LoC). Our improved fuzzer also achieved 26.6$\times$ faster execution and discovered 3.6$\times$ more basic blocks than WinAFL, the state-of-the-art fuzzer on Windows. By fuzzing these 59 harnesses, WINNIE successfully found 61 bugs from 32 binaries. Out of the 59 harnesses, WinAFL only supported testing 29 binaries.

4.2 Background: Why Harness Generation?

Fuzzing is a popular automated technique for testing software. It generates program inputs in a pseudo-random fashion and monitors program executions for abnormal behaviors (*e.g.*, crashes, hangs or assertion violations). Since it was introduced, fuzzing has found tens of thousands of bugs [108].

Most popular fuzzers employ *greybox, feedback-guided* fuzzing. Under this paradigm, fuzzers treat programs like black boxes, but also rely on light-weight instrumentation techniques to collect useful feedback (*e.g.*, code coverage) from each run. The feedback is used to measure how an input helps explore the program's internal states. Thus, a fuzzer can gauge how effective an input is at eliciting *interesting* behaviors from the program.

Fuzzer	AFL	WinAFL	HonggFuzz	Peach	WINNIE
Feedback	✔	✔	✗	✗	✔
Forkserver	✔	✗	✗	✗	✔
Open-source	✔	✔	✔	✗	✔
Windows	✗	✔	✔	✔	✔

Table 4.1: Comparison between various Windows fuzzers and Linux AFL. We compare several key features that we believe are essential to effective fuzzing. WINNIE aims to bring the ease and efficiency of the Linux fuzzing experience to Windows systems.

Intuitively, since most bugs lie in the relatively complicated parts of code, the feedback guides the fuzzer towards promising parts of the program. This gives greybox fuzzers a decisive advantage over black-box fuzzers which blindly generate random inputs without any runtime feedback.

AFL [2], a popular Linux fuzzer, exemplifies greybox fuzzing in practice. Figure 4.2 depicts AFL's fuzzing process. The testing process is similar to a genetic algorithm. It proceeds iteratively, mutating and testing new inputs each round. Inputs which elicit bugs (*i.e.*, crashes or hangs) or new code coverage from the program are selected for further testing, while other uninteresting inputs are discarded. Across many cycles, AFL learns to produce interesting inputs as it expands the code coverage map. Although simple, this strategy is surprisingly successful: several recent advanced fuzzers [96, 95, 41] follow the same high-level process. Overall, AFL-style, greybox fuzzing has proven extremely successful on Linux systems.

Although most recent research efforts focus on improving fuzzing Linux applications [2, 95, 96, 3, 109, 41, 37], Windows programs are also vulnerable to memory safety issues. Past researchers have uncovered many vulnerabilities by performing a manual audit [13]. In fact, Windows applications are especially interesting because they are commonly used on end-user systems. These systems are prime targets for malicious attackers [16, 14]. Automatic Windows testing would pave a way for researchers to look for bugs in many Windows programs while limiting manual code review. In turn, this would help secure the Windows ecosystem.

Unfortunately, no fuzzers can test Windows applications as effectively as AFL can test Linux applications. Table 4.1 compares Linux AFL with popular Windows fuzzers. WinAFL is a fork of AFL ported for Windows systems [110] and supports feedback-driven fuzzing. HonggFuzz supports Windows, but only for fuzzing binaries in dumb mode, *i.e.*, without any coverage feedback [111]. Peach is another popular fuzzer with Windows support but requires users to write specifications based on their knowledge of the fuzzed program [6]. Overall, although there are several rudimentary fuzzers for Windows systems, we find that none offers fast and effortless testing in practice. In this thesis, we aim to address these concerns and make Windows fuzzing truly practical. To do so, we must first examine what the major obstacles are.

4.2.1 The GUI-Based, Closed-Source Software Ecosystem

Compared to Linux programs, Windows applications have two distinguishing features: closed-source and GUI-based. First, many popular Windows applications are commercial products and thus closed-source, like Microsoft Office, Adobe Reader, Photoshop, WinRAR, and Visual Studio. As these commercial applications contain proprietary intellectual property, most of them are very unlikely to be open-sourced in the future. Second, Windows software is predominantly GUI-based. Unlike on Linux which features a rich command-line experience, essentially all of the aforementioned Windows programs are GUI applications. Due to the closed nature of the ecosystem, vendors rarely have an incentive to provide a command-line interface, as most end-users are most familiar with GUIs. In other words, the only way to interact with most programs' core functionality is through their GUI.

GUI applications pose a serious obstacle to effective fuzzing. First, GUI applications typically require user interaction to get inputs, and cannot be tested automatically without human intervention. Bypassing the GUI is nontrivial: it is slow to fully automate Windows GUIs with scripting [99]; meanwhile avoiding the user interface altogether usually requires a deep understanding of the application's codebase, as programmers often intertwine the

Program	Harness	GUI	Ratio	Program	Harness	GUI	Ratio
HWP-jpeg	117	4075	34.8×↑	Tiled	28	720	25.7×↑
Gomplayer	15	1105	73.6×↑	ezPDF	184	4397	23.8×↑
ACDSee	16	510	31.8×↑	EndNote	30	1461	23.8×↑

Table 4.2: Execution times (ms) with and without GUI. GUI code dominates fuzzing execution time (35× slower on average). Thus, fuzzing harnesses are crucial to effective Windows application fuzzing. We measured GUI execution times by hooking GUI initialization code.

asynchronous GUI code with the input processing code [112]. Second, GUI applications are slow to boot, wasting a lot of time on GUI initialization. Table 4.2 shows the startup times of GUI applications compared to a fully-CLI counterpart. In our experiments, GUI code often brought fuzzing speeds down from 10 or more executions per second to less than one. Naturally, fuzzing a CLI version of the application is absolutely essential. WinAFL [110] acknowledges this issue, and recommends users to create fuzzing harnesses.

4.2.2 Difficulty in Creating Windows Fuzzing Harnesses

It is a common practice to write fuzzing *harnesses* to test large, complicated software [100, 101]. In general, a harness is a relatively small program that prepares the program state for testing deeply-embedded behaviors. Unlike the original program, we can flexibly customize the harness to suit our fuzzing needs, like bypassing setup code or invoking interesting functions directly. Hence, harnesses are a common tactic for enhancing fuzzing efficacy in practice. For instance, Google OSS-Fuzz [113] built a myriad of harnesses on 263 open-source projects and found over 15,000 bugs [108].

Harnesses are especially useful when testing GUI-based Windows applications. First, we can program the harness to accept input from a command-line interface, thus avoiding user interaction. This effectively creates a dedicated CLI counterpart for the target program which existing fuzzers can easily handle. Second, using a harness avoids wasting resources on GUI initialization, focusing solely on the functionality at the heart of the program (*e.g.*, file parsing) [102, 103, 105].

Attributes	Fudge	FuzzGen	Winnie
Binary	✗	✗	✔
Target OS	Linux	Linux/Android	Windows
Control-flow analysis	✔	✔	✔
Data-flow analysis	✔	✔	✗
Input analysis	Heuristic	-	Dynamic trace
Ptr / Struct analysis	Heuristic	Value-set analysis	Heuristic

Table 4.3: Comparison of harness generation techniques. Most importantly, WINNIE supports closed-source applications by approximating source-level analyses. Fine-grained data-flow tracing is impractical without source code as it incurs a large overhead.

Unfortunately, Windows fuzzing faces a dilemma: due to the nature of the Windows ecosystem, effective fuzzing harnesses are simultaneously indispensable yet very difficult to create. In addition, due to the prevalence of closed-source applications, many existing harness generation solutions are inadequate [100, 101]. As a result, harness creation often requires in-depth reverse engineering by an expert, a serious human effort. In practice, this is a serious hindrance to security researchers fuzzing Windows applications.

Fudge and FuzzGen. Fudge [100] and FuzzGen [101] aim to automatically generate harnesses for open-source projects. Fudge generates harnesses by essentially extracting API call sequences from existing source code that uses a library. Meanwhile, FuzzGen relies on static analysis of source code to infer a library's API, and uses this information to generate harnesses. Table 4.3 highlights the differences between the existing solutions and WINNIE. Most crucially, Fudge and FuzzGen generally target *open-source* projects belonging to the *Linux* ecosystem, but WINNIE aims specifically to fuzz *COTS, Windows* software. Although it may seem that Linux solutions should be portable to Windows systems, the GUI-based, closed-source Windows software ecosystem brings new, unique challenges. As a result, these tools cannot be used to generate harnesses for Windows applications.

Fudge, FuzzGen, and WINNIE all employ heuristics to infer API control-flow and data-flow relationships. However, whereas Fudge and FuzzGen can rely on the availability of source code, WINNIE cannot as a large amount of API information is irrevocably destroyed

during the compilation process, especially under modern optimizing compilers. Thus, although Fudge and FuzzGen's analyses are more detailed and fine-grained, they are crucially limited by their reliance on source code. This is the fundamental reason why these existing solutions are not applicable to Windows fuzzing. Hence, a new set of strategies must be developed to effectively generate fuzzing harnesses *in the absence of source code*.

4.3 Challenges and Solutions

WINNIE's goal is to automate the process of creating fuzzing harnesses in the absence of source code. From our experience, even *manual* harness creation is complicated and error-prone. Thus, before exploring automatic harness generation, we will first discuss several common difficulties researchers encounter when creating harnesses manually.

4.3.1 Complexity of Fuzzing Harnesses

Fuzzing harnesses must replicate all behaviors in the original program needed to reach the code that we want to test. These behaviors could be complex and thus challenging to capture in the harness. For instance, a harness may have to initialize and construct data structures and objects, open file handles, and provide callback functions. We identified four major steps to create a high-quality harness: ① target discovery; ② call-sequence recovery; ③ argument recovery; ④ control-flow and data-flow dependence reconstruction.

To illustrate these steps in action, we look into a typical fuzzing harness, shown in Figure 4.3. XnView is an image organizer, viewer and editor application [97]. Although the original program supports more than 500 file formats [114], our goal is to test the JPM parser, implemented in the library `ldf_jpm.dll`. Figure 4.3 shows the corresponding harness. First, the harness declares callback functions (lines 2-3), and initializes variables (lines 6 and 9). Second, the harness imitates the decoding logic of the original program: it opens and reads the input file (line 10), retrieves properties (lines 14-17), decodes the image (line 20), and closes it (line 23). Lastly, the harness declares the required variables (line 9) and uses them

```
1  // 1) Declare structures and callbacks
2  int callback1(void* a1, int a2) { ... }
3  int callback2(void* a1) { ... }
4
5  // 2) Prepare file handle
6  FILE *fp = fopen("filename", "rb");
7
8  // 3) Initialize objects, internally invoking ReadFile()
9  int *f0_a0 = (int*) calloc(4096, sizeof(int));
10 int f0_ret = JPM_Document_Start(f0_a0, &callback1, &fp);
11 if (f0_ret){ exit(0); }
12
13 // 4) Get property of the image
14 int f1_a2 = 0, int f4_a2 = 0;
15 JPM_Document_Get_Page_Property((void *)f0_a0[0], 0xA, &f1_a2);
16 ...
17 JPM_Document_Get_Page_Property((void *)f0_a0[0], 0xD, &f4_a2);
18
19 // 5) Decode the image
20 JPM_Document_Decompress_Page((void *)f0_a0[0], &callback2);
21
22 // 6) Finish the harness
23 JPM_Document_End((void *)f0_a0[0]);
```

Figure 4.3: An example harness, synthesized by our harness generator. It tests the JPM parser inside the `ldf_jpm.dll` library of the application XnView. The majority of the harness was correct and usable out of the box. We describe the steps taken to create this harness in §4.3.1 and in more detail in §4.4. Low level details are omitted for brevity.

appropriately (lines 15, 17, 20 and 23). Conditional control flow based on return values is also considered to make the program exit gracefully upon failures (line 11).

❶**Target discovery.** The first step of fuzzing is to identify promising targets that handle user inputs. This process can be time-consuming as, depending on the program, the input may be specified in a variety of ways, such as by filename, by file descriptor, or by file contents (whole or partial). In this example, the researcher should identify that the API `JPM_Document_Start` from `ldf_jpm.dll` library is responsible for accepting the user input through a pointer of an opened file descriptor (line 10).

❷**Call-sequence recovery.** The harness must reproduce the correct order of all function calls relevant to the target library. In this example, there are total 10 API calls to be reconstructed in the full harness. Note that static analysis alone is not enough to discover all callsites. Due to the prevalence of indirect calls and jump tables, researchers must also use

dynamic analysis to get the concrete values of the call targets.

❸**Argument recovery.** The harness must also pass valid arguments to each function call. Reconstructing these arguments is challenging: the argument could be a pointer to a callback function (like `&callback1` at line 10), a pointer to an integer (like `&f1_a2` at line 15), a constant (like `0xa` at line 15), or many other types. When manually constructing a harness, the researcher must examine every argument for each API call, relying on their expertise to determine what the function expects.

❹**Control-flow and data-flow dependence.** It is oftentimes insufficient to simply produce a list of function calls in the right order. Moreover, libraries define implicit semantic relationships among APIs. These relationships manifest in control-flow dependencies and data-flow dependencies. For example, a conditional branch between API calls may be required for the harness to work, like the if-statement at line 11 of the example. Alternatively, one API may return or update a pointer which is used by a later API call. Unless these relationships are respected, the resulting harness will be incorrect, yielding false positives and spurious crashes. For example, the above code updates array `f0_a0` at line 10, and uses the first element in lines 15, 17, 20, and 23. In the absence of source code, this step is extremely challenging, and even the most advanced harness generator cannot guarantee correctness. Human intuition and experience can supplement auto-analysis when reverse-engineering.

4.3.2 Limitations of Existing Solutions

As Windows does not provide fast process cloning machinery (*e.g.*, Linux's `fork`), fuzzers usually start each execution from the very beginning. Considering the long start-up time of Windows applications (see Table 4.2), each re-execution wastes a lot of time to reinitialize the program. Existing solutions (*e.g.*, WinAFL) resort to a technique known as *persistent mode* to overcome the re-execution overhead [73]. In persistent mode, the fuzzer repeatedly invokes the target function in a tight loop *within the same process*, without reinitializing the

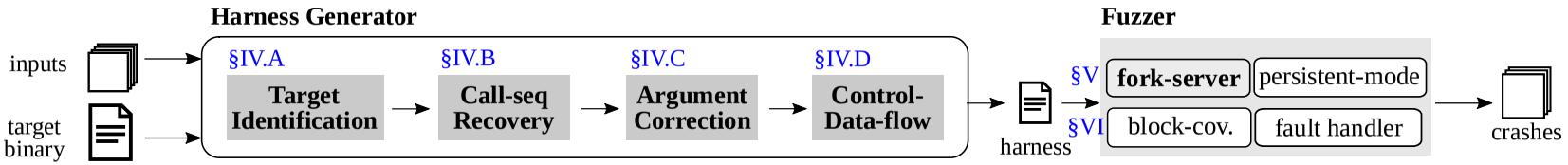

Figure 4.4: Overview of WINNIE. Given the target program and a set of sample inputs, WINNIE aims to find security vulnerabilities. It uses a harness generator to synthesize simple harnesses from the execution trace, and then fuzzes harnesses efficiently with our implementation of `fork`.

program each iteration. To realize the most performance gains, one generally aims to test as many inputs as possible per new process.

While persistent mode partially addresses the performance issues of Windows fuzzing, its efficacy is limited by its strict requirements on the loop body. Specifically, persistent mode expects harnesses to behave like *pure functions*, meaning that harnesses avoid any side-effects, such as leaking memory or modifying global variables. Otherwise, each execution would start from a different program state. Since the harness is repeatedly looped for thousands of iterations, even the smallest side-effects will gradually accumulate over time, finally leading to problems like memory leaks, unreproducible crashes and hangs, and unreliable coverage. For example, a program that leaks 1MB of memory per iteration will reach WinAFL's default memory limit and be terminated. We experienced such errors very often in practice, and discuss more details later in §4.7.1.

Many side-effect errors from persistent mode are difficult to debug or difficult to circumvent. A common issue is that persistent mode cannot continue if the target function does not return to the caller. For example, a program can implement error handling by simply terminating the program. Because most inputs generated during fuzzing are invalid (albeit benign), this still demands constant re-execution, severely degrading performance. Another common problem is that a program will open the input file in exclusive mode (*i.e.*, other processes cannot open the same file) without closing it. This prevents the fuzzer from updating the input file in the next iteration, breaking persistent mode. Problems like these limit the applicability and scalability of persistent mode fuzzers.

4.3.3 Our Solutions

We propose WINNIE, an end-to-end system that addresses aforementioned obstacles to effectively and efficiently fuzz Windows applications. WINNIE contains two components, a harness generator that synthesizes harnesses for closed-source Windows programs with minimal manual effort (§4.4), and a fuzzer that can handle uncooperative target applications

with our efficient `fork` implementation (Figure 4.7). Figure 4.4 shows an overview of our system. Given the program binary and sample inputs, our tracer runs the program and meanwhile, collects dynamic information about the target application, including API calls, arguments and memory contents. From the trace, we identify interesting fuzzing targets that handle user input, including functions in external libraries and locations inside the main binary. For each fuzzing target, our harness generator analyzes the traces and reconstructs related API sequences as a working harness. We test the generated harnesses to confirm their robustness and effectiveness, and then launch fuzzing instances with our fork-server to find bugs. In the following sections, we will use the harness shown in Figure 4.3 as an example to explain the design of each component of WINNIE.

4.4 Harness Generation

To generate the harness, WINNIE followed the four steps previously outlined in §4.3.1. Consider XnView as an example:

❶For *target discovery* (§4.4.1), we trace XnView while opening several JPM files, and then search the traces for input-related APIs, such as `OpenFile` and `ReadFile`.

❷For *call-sequence recovery* (§4.4.2), we search the traces for function calls related to the fuzzing target. In the example, we find all the function calls related to the chosen library (lines 10, 15, 17, 20 and 23). We put the call-sequence into the harness, forming a *harness skeleton*. The skeleton is now more-or-less a simple series of API calls, which we then flesh out further.

❸For *argument recovery* (§4.4.3), we analyze the traces to deduce the prototype for each function in the call sequence. The traces contain verbose information about APIs between the main binary and libraries, like arguments and return values.

❹Finally, we establish the *relationships* (§4.4.4) among the various calls and variables presented in the harness skeleton and emit the final code after briefly *testing* (§4.4.5) the candidate harness. WINNIE also points out complicated logic potentially missed by our

Class	Type	What to record
① **Module**	string	name, path, module
② **Call/Jump**	inter-module	thread id, caller, callee, symbols, args
	intra-module	same as above, only for main .exe
③ **Return**	inter-module	thread id, callee, caller, retval
	intra-module	same as above, only for main .exe
④ **Arg/RetVal**	constants	concrete value
	pointers	address and referenced data (recursively)

Table 4.4: Dynamic information collected by the tracer. We record detailed information about every inter-module call. We also record the same information for intra-module calls within the main binary. If the argument or return value is a pointer, we recursively dump memory around the pointed location. We then use this information to construct fuzzing harnesses (§4.4).

tracer (such as the callback function at line 20) as areas for further improvement.

4.4.1 Fuzzing Target Identification

In this step, WINNIE evaluates whether the program can be fuzzed and tries to identify promising target functions. We begin by performing dynamic analysis on the target program as it processes several chosen inputs. Table 4.4 shows a detailed list of items that the tracer captures during each execution. ① We record the name and the base address of all loaded modules. ② For each call and jump that transfers control flow between modules, our tracer records the current thread ID, the caller and callee addresses, symbols (if available), and arguments. Without function prototype information, we conservatively treat all CPU registers and upper stack slots as potential arguments. ③ We record return values when encountering a return instruction. ④ If any of values fall into accessible memory, we conservatively treat it as a pointer and dump the referenced memory for further analysis. To capture multi-level pointer relationships (*e.g.*, double or triple pointers), we repeat this process recursively. For pointers, we also recognize common string encodings (*e.g.*, C strings) and record them appropriately.

Using our captured traces, we look for functions which are promising fuzzing targets. It is commonly believed that good fuzzing targets have two key features [115, 73]: the library accepts the user-provided file path as the input, and it opens the file, parses the content and

closes the file. We use these two features to find candidate libraries for fuzzing. Specifically, for each function call, we check whether one of its arguments points to a file path, like `C:\my_img.jpm`. To detect user-provided paths, our harness generator accepts filenames as input. Next, we identify callers of well-known file-related APIs such as `OpenFile` and `ReadFile`. If a library has functions accepting file paths, or invokes file-related APIs, we consider it is an input-parsing library and treat it as a fuzzing candidate.

WINNIE also identifies library functions that do not open or read the file themselves, but instead accept a file descriptor or an in-memory buffer as input. To identify functions accepting input from memory, our tracer dissects pointers passed to calls and checks if the referenced memory contains any content from the input file. We also verify that the appropriate file-read APIs were called. To find functions taking file descriptors as inputs, we inspect all invocations of file-open APIs and track the opened file descriptors. Then, we check whether the library invokes file-related APIs on those file descriptors.

Our harness generator focuses primarily on the external interfaces a library exposes. On the other hand, we do not record control flow within the same module as these represent libraries' internal logic. Because invoking the API through those interfaces models the same behavior as the original program, inter-module traces are sufficient for building an accurate harness. However, we treat the main executable as a special case and record all control-flow information within it. This is because the main executable is responsible for calling out to external libraries. Thus, we also search the intra-module call-graph of the main executable for suitable fuzzing targets.

WINNIE then expands its search to within the main binary by analyzing its call-graph. Specifically, WINNIE begins at the *lowest common ancestor (LCA)* of I/O functions and the parsing library APIs we previously identified. In a directed acyclic graph, the LCA of two nodes is the deepest one that can reach both. In our case, we search for the lowest node in the main binary's callgraph that satisfies two criteria. First, it should be before the file-read operation so that our fuzzer can modify the input. Note that even if the fuzzed process has

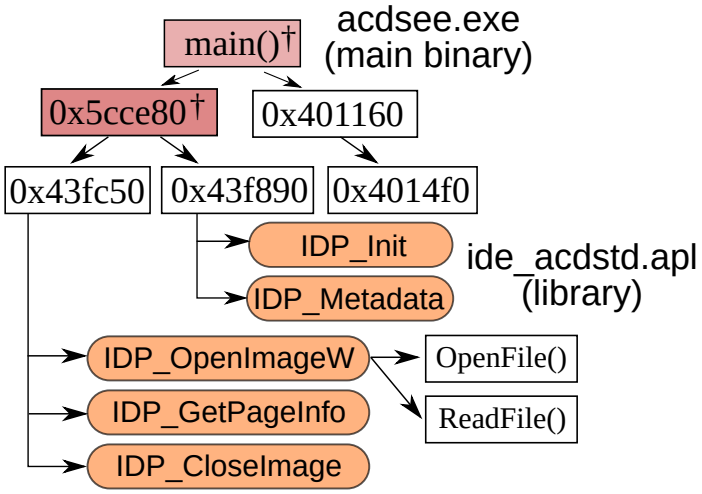

Figure 4.5: A simplified call-graph of the ACDSee program. WINNIE analyzes the call-graph for fuzzing possible targets, focusing on inter-module calls and I/O functions. We look for functions that can reach both I/O functions and also the interesting ones we wish to fuzz. "†" indicates such functions, known as *LCA candidates* (§4.4.1).

opened the input file, we still can modify it so the program uses the new content. Second, the LCA should reach locations that invoke parsing functions. Figure 4.5 shows an example callgraph from the program ACDSee. The function at address `0x5cce80` is the LCA as it reaches two file-related APIs (*i.e.*, `OpenFile` and `ReadFile`) and also invokes the parsing functionality in `ide_acdstd.apl`. We also consider the LCA's ancestors (*e.g.*, `main()`) as fallback candidates, if the immediate LCA does not yield a working harness. In cases where a working LCA is found, it often is sufficient for making an effective harness.

Our tool can also optionally use differential analysis to refine the set of candidate fuzzing targets. Given two sets of inputs, one triggering the target functionality and another not triggering, WINNIE will compare the two execution traces and locate the library functions that are specific to the target functionality. We discard the other functions which are present in both sets of traces. This feature helps deal with multi-threaded applications where only one thread performs operations related to the input file. In any case, differential analysis is optional; it only serves as an additional criteria to improve harness generation.

4.4.2 Call-sequence Recovery

Now that we have identified a candidate fuzzing target, our goal in this step is to reproduce a series of API calls which will correctly reach and trigger the functionality we wish to fuzz. We call such an API sequence a *harness skeleton*. We search the traces for function calls related to that library and copy them to the harness skeleton (lines 10, 15, 17, 20, 23 in Figure 4.3). We also reconstruct the functions' prototypes (*e.g.*, argument count and types) with hybrid analysis: we combine the static analysis provided by IDA Pro [116] or Ghidra [117] with concrete information retrieved from the dynamic execution traces. Namely, we apply pointer types to arguments that were valid addresses in the traces, as the static analysis can misidentify pointer arguments as integers. Lastly, we attach auxiliary code that is required to make the harness work, like a `main` function, forward function declarations, and helper code to open or read files (line 6).

Special care must be taken to handle applications which use multiple threads. In that case, we will only consider the threads that invoke file-related APIs. This is to avoid adding irrelevant calls that harm the correctness of the harness. We encountered several programs that exhibit this behavior, such as GomPlayer, which had hundreds of irrelevant function calls in the execution trace. When the program creates multiple threads within the same library, the trace records an interleaving of many threads' function calls combined. However, since we recorded the thread IDs in our previous step, we can untangle the threads to focus on them individually. With the per-thread analysis, we can narrow the number of calls down to just seven.

4.4.3 Argument Recovery

In this step, we reconstruct the arguments that should be passed to each API call in the call sequence recovered in the previous step. WINNIE attempts to symbolize the raw argument values recorded in the traces into variables and constants. First, we identify pointer arguments. We do so empirically through differential analysis of the trace data. Specifically,

the tracer runs the program with the same input twice, both times with address space layout randomization (ASLR) enabled [118]. Because ASLR randomizes memory addresses across different runs, two pointers passed to the same call site will have different, pseudo-random values that are accessible addresses both times. If this is the case, we can infer that the argument is a pointer. For pointer arguments, we use the concrete memory contents from the trace, dissecting multiple levels of pointers of necessary. Otherwise, we simply consider the value of the argument itself.

Next, we determine whether the argument is *static* or *variable*. Values which vary from execution to execution are *variable*, and we define names for variables and replace their uses with new names. Values which remain constant between runs are *static*, and we simply pass them as the constant value seen in the trace (like `0xA` and `0xD` in Figure 4.3).

4.4.4 Control-Flow and Data-Flow Reconstruction

WINNIE analyzes the program to reflect control-flow and data-flow dependencies in the harness. Control-flow dependencies represent how the various API calls are logically related (*e.g.*, the if-statement on line 11 in Figure 4.3). To find control-flow dependencies, we apply static analysis. Specifically, WINNIE analyzes the control-flow between two API calls for paths from the return value of the invoked function to a termination condition (*e.g.*, `return` or `exit()`). If such a path is found, WINNIE duplicates the decompiled control-flow code (*e.g.*, `if`-statements). The current version of WINNIE avoids analyzing complex flows involving multiple assignments or variable operands in the conditional statement; we leave such cases to a human expert. This is important for accurate harness generation: neglecting control-flow dependencies causes incorrect behavior. For example, consider a harness that fails to reflect an early exit error handling condition in the original program. The program under normal execution would terminate immediately, but the harness would proceed onwards to some unpredictable program state. These kinds of mistakes lead to unreproducible crashes (*i.e.*, false positives).

Data-flow dependencies represent the relationships among function arguments and return values. To find data-flow dependencies, WINNIE tries to connect multiple uses of the same variable between multiple call sites (*e.g.*, `f0_a0` in Figure 4.3). We consider the following possible cases:

- **Simple flows from return values.** Return values of past function calls are commonly reused as arguments for later calls. We detect these cases by checking if an argument always has the same value as a past return value. We only do this for whose values exceed a certain threshold. If we connected any frequently observed values (*e.g.*, connect return value `0` as the next argument), we may generate incorrect harnesses; this resolves many common cases where functions return object pointers.
- **Points-to relationships.** Some arguments are retrieved from memory using pointers returned by previous code. For instance, an API may return a pointer, whose pointed contents are used as an argument in a later API call. In the example harness in Figure 4.3, line 23 uses an argument `f0_a0` that is loaded from memory, initialized by the API `JPM_Document_Start`. When we detect these points-to relationships in the trace, we reflect them in the harness as pointer dereferences (*i.e.*, `*p`). WINNIE also supports multi-level points-to relationships (*e.g.*, double and triple pointers), thanks to the tracer's recursive memory dumping.
- **Aliasing.** WINNIE defines a variable if it observes one or more repeated usages. In other words, if the same non-constant value is used twice as an argument, then the two uses are considered aliases forming a single variable.

4.4.5 Harness Validation and Finalization

Although it covers most common cases, WINNIE's harness generator is not foolproof. WINNIE points out parts of the harness that is unsure about and provides suggestions to help users further improve it. ① We report distant API calls where the second API's call site is far from the first. In such cases, our API-based tracer might have missed some logic between

two API calls. ② We highlight code pointer arguments to users, which could represent callback function pointers or virtual method tables. ③ We provide information about file operations as they are generally important during harness construction.

Once a fuzzing harness has been generated, we perform a few preliminary tests to evaluate its effectiveness. First, we check the harness's stability. We run the harness against several normal inputs; if the harness crashes, we immediately discard it. Second, we evaluate the harness's ability to explore program states. Specifically, we fuzz the harness for a short period and check whether the code coverage increases over time. We discard harnesses that fail to discover new coverage. Lastly, we test the execution speed of the harness. Of all stable, effective harnesses, we present the fastest ones to the user.

WINNIE's goal is to generate harnesses automatically. However, the general problem of extracting program behaviors from runtime traces without source code is very challenging so there will always be cases it cannot cover. Thus, we aim to handle most common cases to maximize WINNIE's ability to save the human researcher's time. We observe that in practice it produces good approximations of valid harnesses, and most of them can be fuzzed with only minor modifications as shown in Table 4.7. We discuss our system's limitations and weaknesses in §4.7.3 and §4.8.

4.5 Fast Process Cloning on Windows

Fork indeed exists on Windows systems [119], but existing work fails to provide a stable implementation. To support efficient fuzzing of Windows applications, we reverse-engineered various internal Windows APIs and services and identified a key source of instability. After overcoming these challenges, we were able to implement a practical and robust fork-server for Windows fuzzing. Specifically, our implementation of the Windows fork corrects the problems related to the CSRSS, which is a user-mode process that controls the underlying layer of the Windows environment [120]. If a process is not connected to the CSRSS, it will crash when it tries to access Win32 APIs. Note that virtually every Windows application

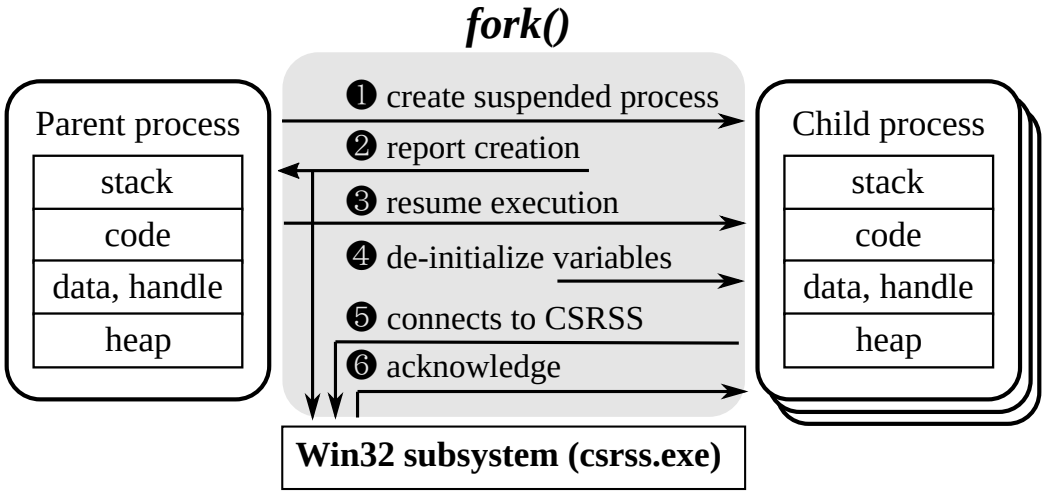

Figure 4.6: Overview of fork() on Windows. We analyzed various Windows APIs and services to achieve a CoW fork() functionality suitable for fuzzing. Note that fixing up the CSRSS is essential for fuzzing COTS Windows applications: if the CSRSS is not re-initialized, the child process will crash when accessing Win32 APIs.

uses the Win32 API. Our fork correctly informs the CSRSS of newly-created child processes, as shown in Figure 4.6. Connecting to the CSRSS is not trivial for forked processes: for the call to succeed, we must manually *de-initialize* several undocumented variables before the child process connects.

To the best of our knowledge, our fork implementation is the only one that can support fuzzing commercial off-the-shelf (COTS) Windows applications. Table 4.5 shows a comparison of process creation techniques on Windows and Linux. `CreateProcess` is the standard Windows API for creating new processes with a default program state, used by WinAFL. New processes must re-execute everything from the beginning, wasting a lot of time on GUI initialization code, shown in Table 4.2. Persistent mode [73] aims to mitigate the re-execution overhead, but is impractical due to the numerous problems outlined in §4.3.2. Thus, our goal is to avoid re-executions altogether by introducing a `fork`-style API. Meanwhile, Cygwin's `fork` implementation is not designed for COTS Windows applications. It works by manually copying the program state after calling `CreateProcess`. It also suffers from problems related to address space layout randomization [121]. The Windows Subsystem for Linux (WSL) is designed for running Linux ELF binaries on Windows.

Fork()	Re-execute	Forkserver				
	CreateProcess	WINNIE	Cygwin	WSL(v1)	WSL(v2)	Linux
Supports PE files?	✔	✔	✔	✗	✗	✗
Copy-on-Write?	✗	✔	✗	✔	✔	✔
Speed (exec/sec)	91.9	310.9	72.8	442.8	405.1	4907.5

Table 4.5: Comparison of `fork()` implementations. Cygwin is not CoW, and WSL does not support Windows PE binaries. WINNIE's new fork API is therefore the most suitable for Windows fuzzing.

Thus, we cannot use it for testing Windows PE binaries, even if it is faster [122]. Our fork implementation achieves a speed comparable to the WSL fork, and most importantly, supports Windows PE applications.

Verifying the Fork Implementation. We ran several test programs under our fork-server to verify its correctness. First, verified that each child process receives a correct copy of the global program state. We checked various the values of global variables in test programs before and after forking a new process. For example, we incremented a global counter in the parent process after each fork and verified that the child process received the old value. Second, to verify that the `fork` implementation is CoW (*copy-on-write*), we initialized large amounts of memory in the parent process before forking. Because the memory footprint of the parent process did not affect the time taken by `fork`, we concluded that our implementation is indeed CoW.

We also briefly measured the speed of `fork` with WinAFL's built-in test program as shown in Table 4.5. On an Intel i7 CPU, we were able to call our `fork` 310.9 times/sec per core with a simple program, which is 4.2× faster than Cygwin's *No-CoW* fork and ∼1.3× slower than the WSL fork. Since we are not using the same fork mechanism as the one provided by the Linux kernel but instead mimicking its CoW behavior using the Windows API, the execution speed is nowhere as fast (*e.g.*, >5,000 execs/sec). Even if Windows implementation of `fork` is slower than Linux's, the time regained from avoiding costly re-executions easily makes up for the overhead of `fork`. Moreover, the process creation machinery on Windows is slow in general: in our experiments, ordinary `CreateProcess`

Category	Component	Lines of code
Harness generator	Dynamic tracer	1.6K LoC of C++
	Synthesizer	2.0K LoC of Python
Fuzzer	Fuzzer	3.0K LoC of C++
	Fork library	0.5K LoC of C++

Table 4.6: WINNIE components and code size

calls (as used by WinAFL) only reach speeds of less than 100 execs/sec. Overall, we believe that the reliability and quality of our Windows fork-server is comparable to ones used for fuzzing on Unix systems.

Idiosyncrasies of Windows Fork. Our `fork` implementation has a few nuances due to the design of the Windows operating system. First, if multiple threads exist in the parent process, only the thread calling `fork` is cloned. This could lead to deadlocks or hangs in multi-threaded applications. Linux's `fork` has the same issue. To sidestep this problem, we target deeply-nested functions that behave in a thread-safe fashion. For example, in the program UltraISO, we bypassed the GUI and fuzzed the target function directly, shown in Table 4.7. Second, handle objects, the Windows equivalent of Unix file descriptors, are not inherited by the child process by default. To address this issue, we enumerate all relevant handles and manually mark them inheritable. Third, because the data structures involved in fork-related APIs differ from version to version of Windows, it is impractical to support all possible installations of Windows. Nevertheless, our fork-server supports all recent builds of Windows 10. Since Windows is very backwards-compatible, we do not see this as a significant limitation of our implementation.

4.6 Implementation

We prototyped WINNIE with 7.1K lines of code (shown in Table 4.6). WINNIE supports both 32- and 64-bit Windows PE binaries. We built our fuzzer on top of WinAFL and implemented the `fork` library from the scratch. The tracer relies on Intel Pin [123] for

dynamic binary instrumentation.

4.6.1 Fuzzer Implementation

Figure 4.7 shows an overview of our fuzzer. We inject a fuzzing agent `agent.dll` into the target program, which cooperates with the fuzzer using a pipe for bidirectional communication. This architecture helps assuage the most uncooperative of fuzzing targets.

The fuzzing agent is injected as soon as the program loads, before any application code has begun executing. Once injected, the agent first hooks the function specified by the harness and promptly returns control to the target application. Then, the target application resumes and initializes itself. The application halts once it reaches the hooks, and the fuzzing agent spins up the fork-server. Since we spin up the fork-server only at some point deep within the program, initialization code only runs once, massively improving performance.

Our fuzzer works as follows: ❶The fuzzing agent, which contains the fork server, is injected into the target application. The injected agent ❷installs function hooks on the entry point and the target function, and ❸instruments all basic blocks so it can collect code coverage. ❹Then, the fuzzer creates forked processes. Using the pipe between the fuzzer and target processes, ❺the agent reports program's status and ❻the fuzzer handles coverage and crash events.

4.6.2 Reliable Instrumentation

Collecting code coverage from closed-source applications is challenging, specially for Windows applications. WinAFL uses two methods to collect code coverage: one using dynamic binary instrumentation using DynamoRIO [110], and another using hardware features through Intel PT (IPT) [124]. Unfortunately, DynamoRIO and IPT are prone to crashes and hangs. In our evaluation, WinAFL was only able to run 26 of 59 targets.

To address this issue, we discard dynamic binary instrumentation in favor of *fullspeed fuzzing* [125] to collect code coverage. Fullspeed fuzzing does not introduce any overhead

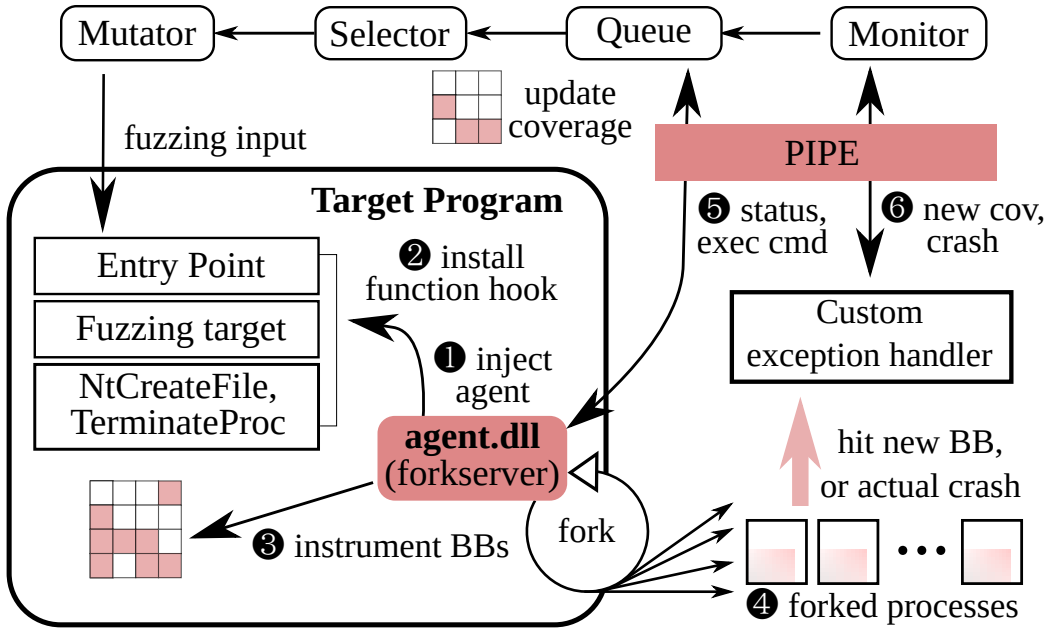

Figure 4.7: Overview of WINNIE's fuzzer. We inject a fuzzing agent into the target. The injected agent spawns the fork-server, instruments basic blocks, and hooks several functions. This improves performance (§4.6.1) and sidesteps various instrumentation issues (§4.6.2).

except when the fuzzer discovers a new basic block. Based on boolean basic block coverage, fullspeed fuzzing only considers there to be new coverage when a new basic block is visited. To implement this, we patch all basic blocks of the tested program with an `int 3` instruction. Then, we fuzz the patched program and wait for the execution to reach a new block. When reached, the first byte of the new block is then restored so that it will no longer generate exceptions in the future. Since encountering new basic blocks is rare during fuzzing, fullspeed fuzzing has negligible overhead and can run the target application at essentially native speed. Breakpoints need only be installed *once* thanks to the fork-server: child processes inherit the same set of breakpoints as the parent. We noticed that this is an important optimization as we observe Windows applications easily contain a massive number of basic blocks (*e.g.*, >100K).

4.7 Evaluation

We evaluated WINNIE on real-world programs to answer the following questions:

- **Applicability of WINNIE.** Can WINNIE test a large variety of Windows applications? (§4.7.1)
- **Efficiency of `fork`.** How efficient is `fork` on versus other modes of fuzzing like persistent mode? (§4.7.2)
- **Accuracy of harness generation.** How effectively can WINNIE create fuzzing harnesses from binaries? (§4.7.3)
- **Finding new bugs.** Can WINNIE discover new program states and bugs from real world applications? (§4.7.4)

Evaluation Setup. Our evaluation mainly compares WINNIE with WinAFL. Other Windows fuzzers either do not support feedback-driven fuzzing (*e.g.*, Peach [6]), or cannot directly fuzz Windows binaries (*e.g.*, HonggFuzz [111]). We configured WinAFL to use basic-block coverage as feedback and used persistent-mode to maximize performance. Our evaluation of WinAFL considers two modes, the DynamoRIO mode (WinAFL-DR) where WinAFL relies on dynamic binary instrumentation, and the PT mode where WinAFL uses the Intel PT hardware feature to collect code coverage. We enlarged the Intel PT ring buffer sizes from 128 kilobytes to 512 kilobytes to mitigate data-loss issues [93]. We performed the evaluation on an Intel Xeon E5-2670 v3 (24 cores at 2.30GHz) and 256 GB RAM. All the evaluations were run on Windows 10, except WinAFL-DR, which was run on Windows 7 as it did not run properly under Windows 10.

Target Program Selection. We generated 59 valid fuzzing harnesses with WINNIE. We ran all 59 programs test the applicability of WINNIE (§4.7.1). For the other evaluations (§4.7.2 to §4.7.4), we randomly chose 15 GUI or CLI applications among the 59 generated harnesses due to limited hardware resources (*i.e.*, 15 apps $\times$ 24 hrs $\times$ 5 trials = 5,400 CPU hrs). We aimed to show that WINNIE can fuzz complicated GUI applications and that WINNIE also outperforms existing solutions on CLI programs. Thus, we chose a mixture of both types of binaries from a variety of real-world applications. For this evaluation, we mainly focused on programs that accept user input from a file, as their parsing components are usually complex

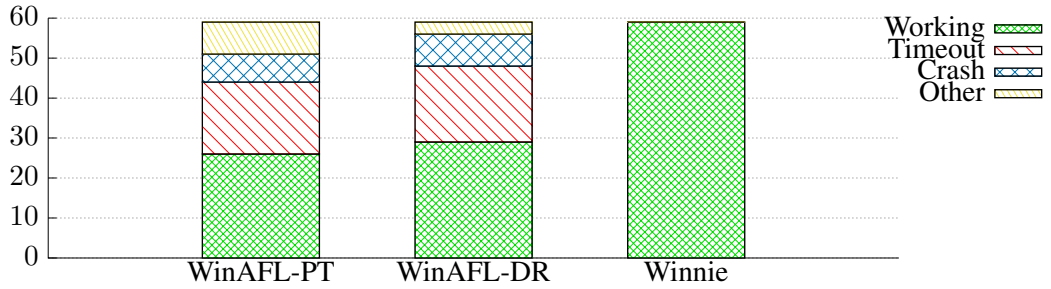

Figure 4.8: Applicability of WINNIE and WinAFL. Among 59 executables, WinAFL-IPT and WinAFL-DR failed to run 33 and 30 respectively, whereas WINNIE was able to test all 59 executables.

(*i.e.*, error-prone) and handle untrusted inputs.

4.7.1 Applicability of WINNIE

Figure 4.8 shows that WINNIE supports running a wider variety of Windows applications than WinAFL. Specifically, WINNIE successfully generates working harnesses for all programs and is able to test them efficiently. WinAFL-IPT failed to run 33 of out 59 harnesses (55.9%) while WinAFL-DR failed to run 30 (50.8%). Execution timeouts during the dry-runs dominate all failed cases of WinAFL (18 for WinAFL-IPT and 19 for WinAFL-DR). Specifically, before the fuzzing fully begins, WinAFL launches a few dry-runs to verify that the fuzzing setup is valid (*e.g.*, harness quality). If the program times out during the dry-run, WinAFL will not be able to continue the testing. The second main failure mode was crashing during the dry-run. This contributed seven failures for WinAFL-IPT and eight for WinAFL-DR. We provide several case studies to understand why WinAFL fails to test these programs:

Unexpected Change in Global State. ① mspdbcmf.exe is a PDB (debug symbol file) conversion tool, and WinAFL failed with a timeout error. When the fuzzer executes the same function iteratively, the program falls into a termination condition, due to a corrupted global variable. In particular, the program assigns a non-zero value to the global variable (`g_szPdbMini`) in the first execution, and the changed value makes the application terminate during the second execution. In other words, the root cause was that the target function

was not idempotent. Unfortunately, WinAFL misclassifies this unexpected termination as a timeout, and thus the fuzzer quits after the dry-run. ② ML.exe (**M**acro assembler and **L**inker) is an assembler program in Visual Studio that crashes when fuzzing begins. Similar to the aforementioned timeout issue, a crash happens at the second execution of the main function. In the first execution, the target program checks the global flag (*i.e.*, `fHasAssembled`) to determine whether the assembly is done and then initializes necessary heap variables. Once the program finishes the first time, it changes the global flag to *true*. In the second execution, the program's control flow diverges because the `fHasAssembled` flag is *true*. This ultimately leads to a crash when it tries to access the uninitialized heap variable.

IPT Driver Issues. The dynamic binary instrumentation adopted by WinAFL-IPT had unknown issues and sometimes prevented WinAFL from collecting code coverage. For example, for the program KGB archiver, we observed that the fuzzer could not receive any coverage due to a Intel-PT driver error.

4.7.2 Benefits of Fork

We tested whether `fork` makes fuzzing more efficient. To do so, we ran the selected programs under our fuzzer in `fork` mode, while we set WinAFL to create a new process for each execution (re-execution mode). Both of these configurations can run the target program reliably. As shown in Table 4.8, `fork` improves fuzzing performance: compared to re-execution mode, WINNIE achieved 31.3× faster execution speeds and discovered 4.0× more basic blocks. In particular, GomPlayer and EndNote recorded 64.7× and 87.3× faster executions and revealed 7.4× and 10.2× more basic blocks respectively.

We also evaluated whether `fork` makes fuzzing more stable. We configured WinAFL to use persistent mode, which runs a specific target function in a loop. Then, we tracked the system's memory and resource usage over time while fuzzing. Almost immediately, we observed memory leaks in the persistent mode harnesses for HWP-jpeg, HWP-tiff, and

Program	Target	Size	API Calls	LoC	Fixed (LoC)	(%)
ACDSee	IDE_ACDStd.apl	3007K	19	506	CB (38), ST (174)	34.3
HWP-jpeg	HncJpeg10.dll	220K	3	92	CB (7), ST (8)	16.3
ezPDF	Pdf2Office.dll	3221K	4	112	CB (2), ST (8)	8.9
HWP-tiff	HncTiff10.dll	630K	3	82	CB (7)	8.5
UltraIso	UltraISO.exe	5250K	1	57	CB (2)	3.5
XnView	ldf_jpm.dll	692K	10	199	CB (4), pointer (2)	3.0
Gomplayer	avformat-gp.dll	4091K	7	116	pointer (2)	1.7
file	magic1.dll	147K	3	96	0	0.0
EndNote	PC4DbLib	2738K	1	55	0	0.0
7z	7z.exe	1114K	1	55	0	0.0
makecab	makecab.exe	50K	1	55	0	0.0
Tiled	tmxviewer.exe	113K	1	55	0	0.0
mspdbcmf	mspdbcmf.exe	1149K	1	55	0	0.0
pdbcopy	pdbcopy.exe	726K	1	55	0	0.0
ml	ml.exe	476K	1	55	0	0.0

CB: Callback function, **ST:** Custom struct

Table 4.7: Harnesses generated by WINNIE. The majority of the harnesses worked out of the box with few modifications. Some required fixes for callback and struct arguments, which we discuss below.

Program	Without Fork				Fork			
	Leak	Hang†	Speed	Cov.	Speed		Coverage	
7z			5.2	1430	49.3	(9.5×↑)	2117	(1.5×↑)
makecab	✗		14.8	576	49.4	(3.3×↑)	1020	(1.8×↑)
GomPlayer		✗	0.4	201	25.9	(64.7×↑)	1496	(7.4×↑)
Hwp-jpeg	✗		4.2	1045	25.9	(6.2×↑)	1847	(1.8×↑)
Hwp-tiff	✗	✗	0.3	1340	26.2	(87.3×↑)	2301	(1.7×↑)
EndNote			5.3	68	89.5	(16.9×↑)	693	(10.2×↑)
Total	3/6	2/6				(**31.3**×↑)		(**4.0**×↑)

Table 4.8: Evaluation of fork(). We ran six applications that both WinAFL and WINNIE could fuzz for 24 hours. We compared their speed and checked for memory and handle (*i.e.*, file descriptor) leaks. `fork` not only improves the performance, but also mitigates resource leaks. Hang† means an execution speed slower than 1.0 exec/sec.

Program	Vendor	Input	GUI?	Size	Speed (exec/sec)			Coverage (# of new BBs)			p-value		Applied heuristics						
					W-DR	W-PT	WINNIE	W-DR	W-PT	WINNIE	W-DR	W-PT	T	L	DF	CS	CB	CF	DF
makecab	Windows 10	.txt	CLI	50KB	228.2	21.3	49.4	762	982	1020	<0.001	<0.001	✔	✔					
HWP-jpeg	Hancom 20	.jpg	GUI	220KB	25.2	21.0	25.9	1821	1498	1847	0.12	<0.001	✔			✔	✔		✔
7z	7-Zip	.7z	Both	1,114KB	8.7	17.0	49.3	1435	1530	2117	<0.001	<0.001	✔	✔					
EndNote	Clarivate	.pdt	GUI	2,738KB	2.1	50.4	89.5	8	37	693	<0.001	<0.001	✔			✔		✔	✔
Gomplayer	GOM Lab	.mp4	GUI	4,091KB	0.2	0.6	25.9	194	1068	1496	<0.001	<0.001	✔		✔	✔			✔
HWP-tiff	Hancom 20	.tif	GUI	630KB	0.2	✗	26.2	1279	✗	2301	<0.001	N/A	✔			✔	✔		✔
Tiled	T. Lindeijer	.tmx	Both	113KB	✗	✗	8.7	✗	✗	36	N/A	N/A	✔	✔					
file	libmagic	.png	CLI	147KB	✗	✗	52.5	✗	✗	116	N/A	N/A	✔	✔					
UltraISO	Ultra ISO	.iso	GUI	5,250KB	✗	✗	45.3	✗	✗	1558	N/A	N/A	✔	✔			✔		
ezPDF	Unidocs	.pdf	GUI	3,221KB	✗	✗	18.9	✗	✗	6355	N/A	N/A	✔			✔	✔		✔
XnView	XnSoft	.jpm	GUI	692KB	✗	✗	23.2	✗	✗	16702	N/A	N/A	✔		✔	✔	✔	✔	✔
mspdbcmf	VS2019	.pdb	CLI	1,149KB	✗	✗	8.1	✗	✗	9637	N/A	N/A	✔	✔					
pdbcopy	VS2019	.pdb	CLI	726KB	✗	✗	28.5	✗	✗	3302	N/A	N/A	✔	✔					
ACDSee	ACDsee	.png	GUI	3,006KB	✗	✗	63.1	✗	✗	618	N/A	N/A	✔		✔	✔	✔	✔	✔
ml	VS2019	.asm	CLI	476KB	✗	✗	44.0	✗	✗	2399	N/A	N/A	✔	✔					

T: Target identification, **L:** LCA, **DF:** Differential analysis, **CS:** Call sequence, **CB:** Callback, **CF:** Control-flow (exit, loop), **DF:** Data-flow (constant/variable, pointer)

Table 4.9: Comparison of WINNIE against WinAFL. Among 15 applications, WinAFL could only run 6, whereas WINNIE was able run all 15. Columns marked "✗" indicate that the fuzzer could not fuzz the application. Markers "✔" indicate which heuristics were applied during harness generation. When both WinAFL and WINNIE support a program, WINNIE generally achieved better coverage and throughput. Although WINNIE excels at fuzzing complicated programs, WinAFL and WINNIE achieve similar results on small or simple programs. We explain in further detail in §4.8. For all other programs, WINNIE's improvement was statistically significant (*i.e.*, p<0.05). P-values were calculated using the Mann-Whitney U test on discovered basic blocks.

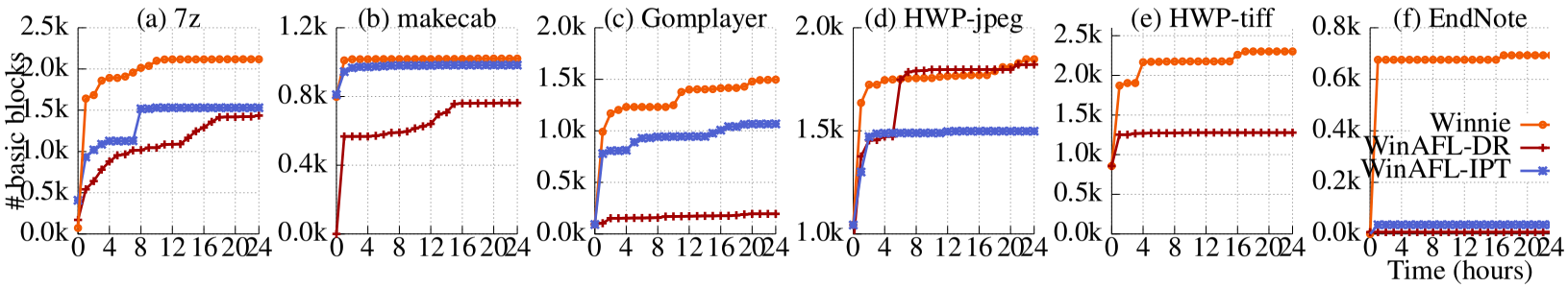

Figure 4.9: Comparison of basic block coverage. We conducted five trials, each 24 hours long, with three fuzzers: WINNIE, WinAFL-DR, and WinAFL-IPT. Only programs which were supported by all fuzzers are shown here; WinAFL was unable to fuzz the rest. When a program can be fuzzed by both WINNIE and WinAFL, their performance is comparable. Nevertheless, most programs cannot be fuzzed with WinAFL.

makecab. The HWP-jpeg and HWP-tiff harnesses also leaked file handles, which would lead to system handle exhaustion if the fuzzer runs for a long time. These types of leaks tend to cause fuzzing to unpredictably fail after long periods of fuzzing, creating a big headache for the human researcher. We explain this in further detail in §4.3.2. `fork` prevented the memory leaks and file handle leaks, improving stability. We further discuss the advantages and disadvantages of persistent mode in §4.8.

4.7.3 Efficacy of Harness Generation

In this section, we evaluate how well WINNIE helps users create effective fuzzing harnesses. To do so, we diffed the initial and final harness code in our evaluation. We analyzed the fixes required to make the harnesses work, and present the findings in Table 4.7 and Table 4.9. As shown, the majority of the harnesses worked with no modifications. On average, the synthesized harnesses had 82.7 LoCs, relied on 3.2 heuristics, and required only 3.4% of the code to be modified. Based on our findings, we discuss the various strengths and weaknesses of the harness generator below.

Strengths of the Harness Generator. The execution tracer provides helpful information about the target program, such as promising fuzzing targets (*i.e.*, Table 4.9: Target identification). This saves the user's time. While creating harnesses, we kept most the original code that WINNIE generated. Without the aid of our system, the user would have had to manually record all of the corresponding function calls and their arguments. The API sequences WINNIE generates also gives useful clues to the user. In the example harness for XnView, since WINNIE extracted 4 calls to the same API with differing arguments, one could conclude that the API's purpose was to initialize various attributes of an object. In our experiments, WINNIE successfully inferred some relationships present in the program (§4.4.4). For example, WINNIE automatically detected that an opened file handle is passed to the next function (lines 6 and 10 in the example Figure 4.3) WINNIE also informs users about constant values, suggesting that they may be magic values that should not be modified.

To assess the usability of WINNIE and its ability to aid human researchers, we recruited two information security M.S. students who were unaware of the project. They were asked to use WINNIE to create fuzzing harnesses for Windows applications of their choice. Within 3 days, they were able to produce 7 functional harnesses, spending roughly only 3 hours per harness on average. The harness generator was most effective when it could rely on a single LCA API (*e.g.*, Table 4.7: 7z). In these cases, the user only needed to collect program run traces and provide them to the harness generator. Upon receiving the trace, WINNIE automatically calculated the LCA and generated C code to correctly invoke the function.

Weaknesses. Although most harnesses worked with few modifications, ACDSee and HWP-jpeg in particular required relatively large modifications (*e.g.*, 34.3% and 16.3% respectively). This is mainly because they passed complex objects and virtual functions to the library's API. One challenge was reconstructing the custom structure layouts without the original source code. Although WINNIE dissects structures and pointer chains from the trace to provide plausible inferences, WINNIE is not perfect. To correct this, we analyzed the object using a decompiler and identified eight variables and four function pointers. Second, we manually extracted the callback functions by adding decompiled code. We followed the function pointers from the trace, and copied the decompiled code into the harness. There will always be some cases that WINNIE cannot handle. We discuss a few examples in §4.8, and we hope to support them in future versions of WINNIE.

4.7.4 Overall Results

Overall Testing Results

Figure 4.9 shows the ability of each fuzzer to find new coverage. Overall, WINNIE discovered 3.6× more basic blocks than WinAFL-DR and 4.1× more basic blocks than WinAFL-IPT. We also applied statistical tests, using p-values to compare the performance of three fuzzers, as suggested by [126]. For WinAFL-DR and WinAFL-IPT, all trials except HWP-jpeg have p-values less than 0.05, meaning that WINNIE's improvement is statistically

Product	Buggy File	Size	Bug Type(s)	Bug(s)
Source Engine	**engine.dll**	6.1M	ND	2
MS WinDBG	**pdbcopy.exe**	743K	Arbitrary OOB read	1
MS Windows	**makecab.exe**	82K	Double free	1
Visual Studio	**ml.exe**	475K	SBOF	1
	undname.exe	23K	SOF	1
Alzip	**Egg.dll**	131K	ND	1
	Tar.dll	114K	Integer underflow	1
	Alz.dll	123K	Stack OOB read	1
Ultra ISO	**ultraISO.exe**	5.3M	Integer overflow, SOF	2
			Uninitialized use	1
XnView	**ldf_jpm.dll**	709K	HC, Integer overflow	2
Hancom Office	**HncBmp10.flt**	85K	Heap BOF	2
	HncJpg,Png,Gif	134-225K	ND	3
	HncDxf10.flt	242K	ND, Integer overflow	3
	HncTif10.flt	645K	HR, TC, FC, HC	6
	IMDRW9.flt	147K	ND, SBOF	2
	ISGDI32.flt	760K	Heap UAF, HC	3
	IBPCX9.flt	83K	Integer overflow, ND	2
FFMpeg	**FFmpeg.dll**†	12.8M	Div by zero	1
Uriparser	**uriparse.exe**†	157K	Integer underflow	1
Gomplayer	**RtParser.exe**	18K	SOF, SBOF, ND	3
EzPDF	**ezPDFEditor.exe**	23.9M	Race condition, ND	3
	Pdf2Office.dll	3.2M	SBOF, SOF, ND	3
VLC player	**Mediainfo.dll**	136K	Integer underflow	1
	libfaad.dll	273K	ND, Denial of service	2
Utable	**Utable.exe**	874K	SBOF	1
RetroArch	**bnes.dll**	2.4M	ND	2
	emux_gb.dll	419K	ND, Div by zero	3
	snes_9x.dll	2.8M	Heap OOB write	1
	quicknes.dll	1.0M	Div by zero	1
Capture2Text	**C2T_CLI.exe**	558K	ND	1
Total	32		19	61

ND: Null-ptr dereference, **HR:** Heap OOB read, **HC:** Heap corruption, **TC:** Type confusion, **FC:** Field confusion, **SOF:** Stack overflow, **SBOF:** Stack buffer overflow

Table 4.10: Bugs found by WINNIE. We discovered total 61 unique vulnerabilities from 32 binaries. All vulnerabilities were discovered on the latest version of COTS binaries. We reported all bugs to the developers. "†" indicates that the bug existed in the released binary, but the developer had already fixed it when we filed our report.

significant.

Real-world Vulnerabilities

WINNIE's approach scales to complex, real-world software. To highlight the effectiveness of our approach, we applied our system to non-trivial programs that are not just large in size but also accompany complicated logic and GUI code. We also included binaries from several well-known open-source projects because most of them have only been heavily fuzzed on Linux operating systems; thus their Windows-specific implementations may still contain bugs. Among them all, WINNIE found 61 previously unknown bugs in 32 binaries (shown in Table 4.10). All these bugs are unique. These bugs cover 19 different types, including but not limited to stack and heap buffer overflow, type confusion, double free, uninitialized use, and null pointer dereference. At the time of writing, we have reported these bugs to their corresponding maintainers and are working with them to help fix the bugs.

4.8 Discussion

Due to the difficulty of fuzzing closed-source, GUI-based applications, most Windows programs are tested either by unscalable manual efforts, or are only evaluated during the development by their vendors. In contrast, Linux programs are consistently tested and improved at all stages of the software lifecycle by researchers over the world. Most prior fuzzing work also has been concentrated on Linux systems. However, as shown in our evaluation, it is easy to find many bugs in Windows software we target—especially given the legacy code bases involved. Nevertheless, we identify several limitations of WINNIE, which can be addressed in the future to better test more programs.

Limitations of Harness-Based Testing. Testing the program with a harness limits the coverage within the selected features. In the case of WINNIE, we cannot reach any code in unforeseen features absent from the trace. Thus, the maximum code coverage possible is limited to the API set the trace covers; the number of generated harness is limited by the number of inputs traced. To mitigate this issue, we recommend users to collect as

many sample inputs as possible to generate a diverse set of harnesses. Although we cannot eliminate this problem inherited from harness-based testing, automatic harness generation will help alleviate the burden of manually creating many harnesses.

Highly-Coupled Programs. It is more challenging for WINNIE to generate harnesses for applications tightly coupled with their libraries. As the logic is split into two binaries, the program may use frequent cross-module calls to communicate, making it hard to accurately identify and extract the relevant code we wish to fuzz. In Adobe Reader, for instance, the main executable `AcroRd32.exe` is simply a thin wrapper of the library AcroRd32.dll [102]. There are a lot of functions calls between these two binaries, or with other libraries, like `jp2.dll`. Thus, the harness generator needs to handle calls between the main executable and a library, callbacks from a library to the main executable, and calls between libraries. Our system focuses on handling cases where the communication merely happens within two components. To support more complicated invocations like in Adobe Reader, we plan to improve our tracer and generator to capture a complete trace of inter-module control- and data-flow.

False Positives. Inaccurate harnesses may generate invalid crashes or exceptions that do not occur in the original program. As a result, WINNIE will mistakenly assume the presence of a bug, leading to a false positive. As described in §4.4.5, WINNIE combats false positives by pre-verifying candidate harnesses during synthesis. Still, eliminating false positives requires a non-negligible effort. Since bug validation must be conducted against the actual application, constructing a suitable input file and interacting with the GUI is required. For example, when fuzzing Adobe Reader's image parser, end-to-end verification requires creating a new PDF with the buggy image embedded, and then opening the image via the GUI. This step can be automated on a per-target basis, and it is mostly an engineering effort. Nevertheless, as long as WINNIE can generate high-quality harnesses, this validation incurs little overhead due to the small number of false crashes.

Focus on Shared Libraries. WINNIE's harness generator focuses testing shared libraries

because shared libraries represent a clear API boundary. Past harness generation work also focuses on testing functions within libraries [100, 101]. Moreover, unlike calls to exported functions in libraries, private functions in the main executable are difficult to extract into independent functions. To fuzz the main binary, we rely on our injected fork-server, allowing any target address in the main binary to be fuzzed.

Performance Versus Persistent Mode. We noticed that WinAFL occasionally shows better performance on certain target applications, typically simple ones. Upon investigation, we found that the performance difference ultimately stems from WinAFL's strong assumptions about the target application. Specifically, WinAFL assumes the harness will not change any global state and will cleanly return back to the caller (§4.3.2). Therefore, it only restores CPU registers and arguments each loop iteration. Instead, WINNIE uses `fork` to comprehensively preserve the entire initialized program state, which incurs a little overhead. However, as shown in the evaluation, our conservative design makes WINNIE support significantly more programs. Although WinAFL performs better on simple programs, it could not test even half of the programs in our evaluation (§4.7.1).

Other input modes. In our evaluation, we focused on fuzzing libraries which accept inputs from files or standard input. Another common way programs accept input is through network packets. WINNIE supports this case. To fuzz these network applications, we extended WINNIE by implementing a *de-socket* [127, 128] technique to redirect socket traffic to the fuzzer.

4.9 Extension: Automatic Generation of Internet Scans for Malware

Remotely accessible applications such as remote access trojans or remote desktop programs are a class of applications that allows an attacker interactive connection to the remote computers. Regardless of their original purpose, these applications have been widely used to leak private information from the remote machine to an attacker; thus rapidly scanning the application and identifying the malicious campaign are the key to protect normal users from

stealing their private data. We extend WINNIE's harness generator for rapid-prototyping malware and extracting the network scanning signature for the two-sided internet scanning and the longitudinal study. To generate the internet scanning signature from any given binary samples, we implemented a prototype of the BLUEPRINT system. First, BLUEPRINT builds a harness program to restore the intended networking behavior such as opening a port and accepting a connection. Second, it retrieves input values that can trigger the data-sending functions (*e.g.*, `send()`). Without the returned message from the sample, the network scanner is not able to classify the program from the connected port.

The main challenge of the internet scanning signature extraction is the uncertainty of the collected samples. Since the dataset is reported and collected without the full packaged information, the typical sample does not provide source code or description about the binary, which demands researchers to conduct heavyweight manual binary analysis to understand the sample's structure. It is also challenging because triggering the intended or hidden networking behavior (*e.g.*, listening a port and accepting a connection) requires an intervention by the user. For an example of the typical remote desktop server which employs GUI, we need to emulate the same user interaction such as clicking the icon or typing text data to enable the server. If the sample requires special conditions like the existence of a specific file or registry, triggering the behavior becomes more difficult. Besides, hidden information in the binary is challenging to discover. For example, the binary can read the socket configuration data (*e.g.*, port number) from its data section and decrypt on-the-fly; thus, typical static analysis is not able to extract the configuration data for extracting the signature.

Definition: Generating Network Scanning Signature. The problem of generating the network scanning signature is defined as: given any collected binaries, extract any payload value to return unique string through the listening socket and its socket configurations that reduce the scanning space. The extracted scanning signature consists of payload, expected return message, and socket configurations.

Research Scope. Analyzing the malicious binary is challenging because the distributor oftentimes attempts to hide their intention by using multiple protection techniques. To limit the scope of our research, we made the following assumptions about the handled sample to be analyzed by BLUEPRINT:

- BLUEPRINT collects samples from any repositories. We do not demand the description or source code of the sample, or dependent library files. Also, we do not selectively collect malware. Instead, we assume that the sample is potentially malicious if it contains port-listening ability.
- Network APIs are used in the sample. Since we are extracting the network scanning signature, the binary should contain the ability to listen to a port and exchange data with an external network.
- The collected sample invokes the imported network APIs and the cross-reference to the that exist in the binary; thus we discard the collected sample if the networking functionality is unreachable (*i.e.*, dead code).
- Since our approach combines both static and dynamic analysis, we consider binary packing and obfuscation as an orthogonal problem with our approach and they require additional or manual analysis; hence, they are not appropriate for large-scale analysis.
- The sample does not require solving a complex cryptographic challenges to retrieve the internet scanning signature.

4.9.1 BLUEPRINT's Methodology

To overcome the aforementioned challenges and extract the network scanning signatures, we employ two methods:

- Network operation restoration: A method to reveal the hidden functions and force to replay the network operation.
- Microscopic symbolic execution: A method to solve the existing constraints to discover the scanning input to send out unique string to the external network.

Method	Pattern	Fuzzing	Sym-exec	Manual	BLUEPRINT
Extract signature	✔	✔	✔	✔	✔
Extract auxiliary	✗	✔	✔	✔	✔
Validation	✗	✔	✗	✔	✔
Scalability	✔	✗	✗	✗	✔

Table 4.11: Comparision between various scanning signature extraction techniques. We compare several key features ath we believe are essential to effective extraction process. BLUEPRINT aims to bring the ease and efficience solution.

The *Network operation restoration* method aims to generate a harness program to run the collected sample without resolving the complicated conditions that the sample contains initially. To be specific, we apply both static and dynamic binary analysis and identify the candidate functions that can reach out to the network APIs. Once we have the candidate function addresses, we execute the target function with proper arguments until we successfully activate the networking routine. To extract the scanning signatures with scale, we utilize *Microscopic symbolic execution* which runs on reachable and relatively short paths. For example, we let the symbolic execution engine works between `recv()` and `send()` if it is reachable from the `recv()`.

To the best of our knowledge, BLUEPRINT is the first approach to automatically extract internet scanning signatures on a large-scale. Nevertheless, various approaches achieved similar goals with BLUEPRINT. String extraction technique [129], fuzzing-based [130], and symbolic execution-based[131] approaches are proposed. Unfortunately, no methods can extract network scanning signatures on a large-scale. Table 4.11 compares BLUEPRINT with other available techniques. String extraction can extract several candidate payloads but does not validate the data on the program. If fuzzing discovers a special input to trigger the actual network operation, it can extract the signature and validate it with the program. However, fuzzing usually requires countless executions; thus cannot tell the success in a short amount of time. Symbolic execution is a good way to reveal the scanning signature with well-defined beginning and termination locations. Unfortunately, it cannot validate the signature because the execution does not operate on the live socket. Although manual

analysis can discover and validate the signature, this method is not applicable for large-scale analysis. In this thesis, we attempt to resolve the aforementioned concerns and make the signature extraction process effective and efficient.

4.9.2 System Architecture

We present BLUEPRINT, an end-to-end system that conducts malware binary analysis to effectively and efficiently extract the internet scanning signatures. BLUEPRINT contains two components, a harness generator that restore an active network operation for scrapping signature and validation, and a signature generator that can resolve constraints to discover particular input to send out unique string to an external network. Figure 4.10 presents an overview of our system. BLUEPRINT accepts collected samples from the repositories (*e.g.*, VirusTotal [132]). Given the sample, the *Harness generator* quickly classifies and decides whether to allocate further resources or discard. Suppose we select the sample for the next step analysis. We generate a harness program that load the sample binary into the virtual memory and calls the specific function (*i.e.*, reachable to network APIs) by leveraging the hybrid binary analysis. From the harness, the *Signature extractor* retrieves scanning signature and auxiliary information (*e.g.*, port number or internet address) from the symbolic executions. Finally, BLUEPRINT returns the scanning rule and launches the internet scanning.

When generating the scanning signature, we follow the four steps:

❶ For *rapid prototype*, we discard any samples if we are not able to apply static binary analysis; thus we check binary packing, customized import address table (IAT), the existence of network API, and cross-reference to the network API.

❷ For *binary analysis*, we reconstruct function call graphs and collect function call paths between any connection control function (*e.g.*, `accept()`) to the data exchange functions (*e.g.*, `recv()` or `send()`). To obtain the necessary information for the harness generation process, BLUEPRINT dynamically analyzes the binary and captures the run traces to acquire

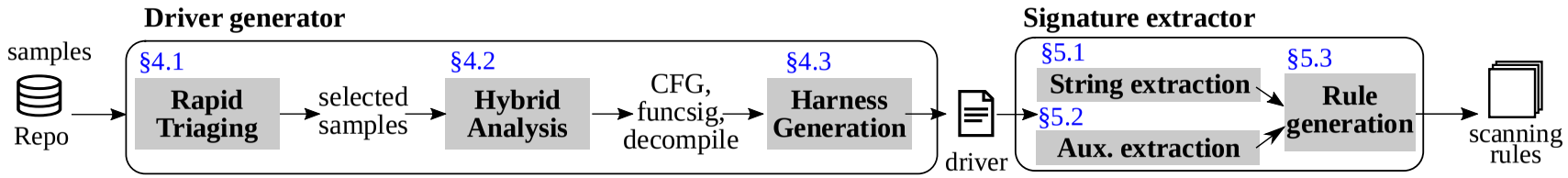

Figure 4.10: Overview of BLUEPRINT. Given the collected sample programs, BLUEPRINT aims to extract internet scanning signatures. It uses a harness generator to synthesize program wrapper from the hybrid binary analysis, and then extract network scanning signatures efficiently with symbolic execution.

the real function call paths and control flow graph. For the signature extraction, BLUEPRINT additionally infers multiple pairs of start/end addresses of the symbolic execution.

❸ For *harness generation*, we infer the least common ancestor (LCA) function of the essential network APIs and call the function with proper arguments.

❹ For *signature extraction*, BLUEPRINT conducts multiple small-scale symbolic executions and generates the scanning signatures.

4.9.3 Network Primitive Restoration

The goal of network primitive restoration is to reproduce the active network operation from the given samples without knowing the actual usage of the binary.

Rapid Prototyping. BLUEPRINT rapidly prototypes the given samples and classify the binaries before engaging the heavy static and dynamic binary analysis because they require a significant amount of H/W resource and time; hence, rapid prototyping is critical for the large-scale application. To do so, BLUEPRINT conducts sequential analysis and quickly classifies binary. First, we check segment names and match those names with well-known binary packer's signature. Second, we enumerate the network-related functions from the import address table and check cross-reference to the functions (*i.e.*, any instruction to call the network API). Finally, BLUEPRINT removes duplicated files. In particular, we de-duplicate the samples with the aid of *imphash* (*i.e.*, a hash of library/API names).

Hybrid Binary Analysis. Now that we have collected analyzable and unique samples, our goal in this stage is to provide sufficient internal information by using hybrid analysis, which is a critical process for harness generation, extracting signature, and extracting the auxiliary information. Once we receive the results from the static analysis, then we fix any misidentified data with solid information collected from the dynamic execution trace.

Table 4.12 shows the list of items that BLUEPRINT demands for further operations. We first enumerate boundary information of all basicblocks and functions as well as their signatures and decompiled code (①, ②, and ⑤). Then, we reconstruct call graph of all

Class	S/D†	Type	What to analyze and collect
①Basicblock	S	address	start, end address pairs
②Function	S	signature	address, name, argument types
③Call path	S	address	call path to the network APIs
④Basicblock path	S	address	basicblock path for each call path address
⑤Decompile	S	code	decompiled code for each referenced function
⑥Sym-exec	S	address	start, end address pairs for symbolic execution
⑦hton/inet_addr	S	data	inferred port numbers and internet addresses
⑧Loaded module	D	data	list of loaded modules and addresses
⑨Run trace	D	address	observed call path and basicblock addresses

†: Static or Dynamic binary analysis

Table 4.12: Collected information during the hybrid binary analysis. BLUEPRINT runs static analysis first and applies the dynamic analysis if the applied heuristics requires dynamic run trace. To reduce the collected volume, BLUEPRINT calculate function call and basicblock paths to the network API and collects the auxiliary information if it belongs to the paths.

Name	Description
①LCA	Infer the least common ancestor functions and call them
②force_init	Initialize the socket on behalf of the sample
③socket_arg	Provide active socket as a function's argument
④infer_arg	Provide proper arguments to minimize error
⑤fake_lib	Provide crafted library to avoid dependency error
⑥before_crash	Run before program makes crash and try other heuristics
⑦force_exec	Force the execution toward the network APIs

Table 4.13: Heuristics used for the harness generation.

functions to extract call paths to the network APIs (③) and calculate the basicblock paths for each referenced function in the call paths (④). For the symbolic execution, BLUEPRINT retrieves pair of start and end addresses, and we infer the expected port number and internet address by checking the argument of corresponding functions (⑥ and ⑦).

Harness Generation. A wrapper program, namely harness, makes the collected sample alive and activates the intended network operation. By doing so, BLUEPRINT can directly extract signatures from the activated port or validate signatures among the candidates proposed by the symbolic execution; thus this is one of the key components. We automatically apply a set of rules and return a C code to generate a harness program. Table 4.13 shows the heuristics that we empirically found very useful:

- **Call LCA function**: We load the target binary and call the LCA function to trigger the hidden behavior.
- **Forced socket initialization**: we first initialize the socket in the harness and then invoke the LCA function for configuring and controlling the socket.
- **Active socket as argument**: If we identify a function that requires listening or accepting the socket, we handle the socket initialization and pass the activated socket as an argument.
- **Safe argument**: When we pass a constant or pointer to the function, we allocate memory (*e.g.*, heap) and fill with the value because the passed variable could be referenced with an offset; thus allowing access to the nearby address will decrease the error ratio.
- **Fake library generation**: BLUEPRINT examines the import address table and reconstructs the fake library file with all function names included.
- **Run sample until crash**: We patch the crash-inducing instruction and let the execution stop before the crash. Then, we invoke the LCA function.
- **Forced execution**: If the execution does not visit the path that we expected but failed to call the network APIs, we patch the binary to force the execution to the intended direction.

4.9.4 Network Scanning Signature Extraction

We use symbolic execution to extract any string or character array for the scanning signature. During the extraction, we do both intra- and inter-procedural symbolic executions. For the intra-procedural symbolic execution, BLUEPRINT retrieves the partial LCA addresses which can reach to the specific network APIs internally. For example, the LCA function address contains APIs for connection control and data exchange. Knowing the function and its corresponding APIs, BLUEPRINT discovers several candidate address pairs for the symbolic execution. If we confirm that the intra-procedural execution is not available, then

Category	Component	Lines of code
Binary analyzer	Triaging	1.0K LoC of Python
	Static analysis	1.2K LoC of Python
Driver generator	Dynamic analysis	0.3K LoC of C++
	Builder	2.3K LoC of Python
Signature extractor	Symbolic executor	0.3K LoC of Python
	Rule gen & Scanner	0.8K LoC of GoLang

Table 4.14: BLUEPRINT components and code size

we seek a chance to the inter-procedural execution. To do so, we choose another LCA address that acrosses multiple callees to the network APIs, and then locates the address pairs. After the symbolic execution, if we discover the solution to resolve the existing constraints, retrieving the input value and expected output is a trivial process. Since all network APIs have a clear description of their arguments, we can directly read the memory data from the end state. For example, if our execution ends at `send()` function, we can access the memory on the second argument, which is the address of the buffer, and finally extract the string.

4.9.5 Prototype Implementation and Preliminary Evaluation

We evaluated BLUEPRINT on collected samples to answer the following evaluation questions:

- **Effectiveness and efficiency of BLUEPRINT:** Can BLUEPRINT apply techniques to a large variety of malicious applications? How efficient is BLUEPRINT in processing large datasets? (§4.9.5)
- **Extracted scanning signature:** Can BLUEPRINT discover the ready-to-use scanning signature? (§4.9.6)

Implementation. We prototyped BLUEPRINT with 5.9K lines of code as shown in Table 4.14. BLUEPRINT handles both 32- and 64-bit PE binaries running on Windows OS. Statis analysis relies on IDA [116] and dynamic analysis conducts binary instrumentation on top of DynamoRIO [79]. BLUEPRINT adopts the ANGR platform [133] for the sym-

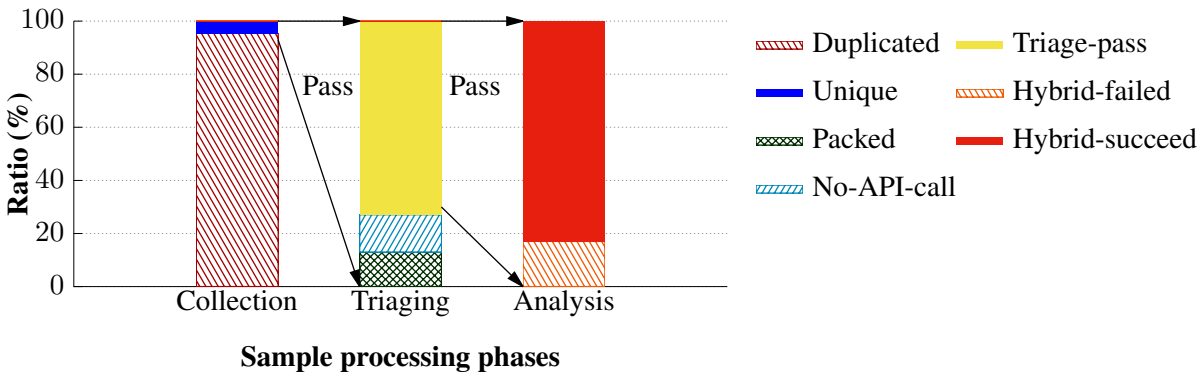

Figure 4.11: Filtered samples for each phase. Upon the sample acquisition, BLUEPRINT passes the de-deplicated files for the triaging. After removing packed and challenging files due to the unclear API paths, BLUEPRINT starts the hybrid analysis.

Driver generator									Signature extractor		
Overall		Applied heuristics (%)									
EXE	DLL	①	②	③	④	⑤	⑥	⑦	#	Payload	Response	Port
10/80	6/20	80%	45%	10%	60%	10%	25%	35%	3.1	3/100	3/100	7/100

Table 4.15: Effectiveness of the replayer and signature extractor. We ran BLUEPRINT on 100 unique samples. Overall, BLUEPRINT was able to enable the port-listening on 20 samples. "Applied heuristics" indicates the ratio of used heuristics and the last column shows the average number of heuristics for individual sample. "Signature extractor" shows the number of succeeded symbolic execution including constraint solving and concretization.

bolic execution and implements the rule generator and internet scanner with Golang to be compatible with ZMAP [134].

Evaluation Setup. Our evaluation mainly checks how many samples can BLUEPRINT handle and how many internet scanning signatures are extracted. First, we configure the YARA rule to collect any submitted samples if they have any network APIs to open a port. Then, BLUEPRINT runs the harness generator and signature extractor to generate the network scanning signatures. We conducted the evaluation on an Intel Xeon(R) E5-2670 v3 (24 cores at 2.30GHz) machine with 256 GB RAM. For the initial triaging, hybrid analysis, and harness generation, we run the evaluation on Windows 10 virtual machine. For the symbolic execution and the internet scanning, we run on Ubuntu 18.04.

Effectiveness and Efficiency of BLUEPRINT

First, we evaluate how well BLUEPRINT extracts internet scanning signatures effectively and efficiently. To do so, we traced how individual sample is classified, passed, or discarded for each stage.

Rapid-Prototyping. The purpose of the rapid-prototyping is to discard any samples that BLUEPRINT is not able to handle and preserve the available resource for other passed binaries. Figure 4.11 shows the performance of our rapid-prototyping process and the ratio of the filtered binaries for each stage. We initially collected 10,000 samples and de-duplicated 95.4% of files, which remained to 460 files. Among the 460 samples, we launched the BLUEPRINT's triaging component. During the classification, BLUEPRINT applied several pattern matching to identify packed binaries and analyzed call instructions to the imported network APIs to find "challenging-to-analyze" binaries. During the triaging process, BLUEPRINT passed 73% of files from the de-duplicated samples (total of 336 binaries) to the next stage (*i.e.*, hybrid analysis) after removing 13% of packed binaries and 14% of unapplicable binaries.

Hybrid Analysis. Before the actual harness generation, hybrid analysis plays a key role for both harness generation and symbolic execution by providing all necessary data to each component from the one-time analysis. After accepting the triaged binaries, the following hybrid analysis discovers the function call and basicblock execution paths to the network APIs as much as possible. Figure 4.11 shows the success ratio and Table 4.15 displays the detailed result of the harness generator and signature extractor. Among all samples from the triaging, hybrid analysis succeeded to finish the mission on 83% of binaries.

Harness generation and signature extraction. BLUEPRINT's harness generator successfully opened a port for 14% of EXE files and 25% of DLL files. As shown in Table 4.15, all seven heuristics were used during the harness generation process and an average of 3.1 heuristics was used for all samples. Among all heuristics, function-LCA (heuristic-①)

and argument inference (heuristic-④) were most frequently used, recording 80% and 60% respectively. Also, symbolic execution succeeded in extracted the signature about the string and port number. On average, symbolic execution extracted meaningful information from 5% of the binaries.

The efficiency of the harness and signature generation. Since BLUEPRINT aims to support large-scale signature generation, it should pass all pipelines quickly without using many resources. Table 4.16 shows the amount of data collected and elapsed time for processing the samples. Compare to all existing functions and basicblocks, BLUEPRINT partially collects necessary information only. For example, when the binary size is less than 1MB, BLUEPRINT collected an average of 3.0 paths for the string extractor and 25.6 function information (*e.g.*, decompiled code and signature) that belong to the path. It means that BLUEPRINT collects 1.1% of function information from the entire binary.

BLUEPRINT spent 363.7 seconds processing one sample on average. In particular, hybrid analysis required 175.2 seconds, and symbolic execution required 92.9 seconds, which accounts for 48.1% and 25.5% of the total execution time respectively. Note that the average time does not correspond to all samples collected. For example, as shown in Figure 4.11, many samples are de-duplicated or discarded during the triaging stage. In this case, the discarded samples consume time for calculating *imphash* or triaging (see the "triaging" column in Table 4.16).

4.9.6 Extracted Signatures and Validation

BLUEPRINT's approach scales to large-scale analysis. To highlight the applicability, we applied BLUEPRINT to randomly collected samples and attempted to extract the scanning signature. Table 4.17 shows the 23 signatures for the internet scanning. Among the signatures, generated harness discovers 18 signatures and symbolic execution discovers 5 signatures. All these signatures are unique and applicable to different binary families.

Filesize	Overall†		Driver		Extractor		funcsig	Avg. processing time (sec)					Data size
	Funcs	BBs	callpath	BB path	string	hton		Triaging	Hybrid	Replayer	Sym-exec	Total	
<1 MB (29.4%)	2.2K	22.5K	3.0	6.5	6.7	12.4	25.6	7.6	42.7	30.7	90.1	171.1	319K
1-5 MB (44.1%)	7.2K	74.3K	10.4	7.3	5.4	19.2	40.7	28.4	213.5	67.5	111.4	420.8	767K
>5 MB (26.5%)	15.7K	184.0K	2.7	6.6	6.0	22.5	33.4	75.0	269.4	77.8	77.2	499.4	815K
Average	8.4K	93.6K	5.4	6.8	6.0	18.0	33.2	37.0	175.2	58.7	92.9	363.7	633.6K

Overall†: Existing information in binary. BLUEPRINT collects necessary data only.

Table 4.16: Efficiency of BLUEPRINT for each stage. We ran the evaluation on three groups divided by the file size. "Overall" indicates the total number of existing functions and basicblocks from our static analysis. "Replayer" and "Extractor" show the number of discovered paths (*e.g.*, call or basicblock) to the interesting APIs. "Avg. processing time" indicates actual time taken for each step.

Family	Class†	sha1[0:5]	Payload	Response	Port
CobaltStrike	malware	9d7376	**HEX**:|0D|0A|	**STR**:HTTP/1.1	-
Farfli	malware	3150a5	**STR**:CONNECT	**STR**:HTTP/1.0	4000-5000
HiddenCobra	malware	e3d038	**STR**:ANY_STR	**STR**:bhjo.org	-
Slingshot	malware	87a28a	**STR**:ANY_STR	**STR**:|B2|7F|23|	-
Mydoom	malware	000549	**STR**:ANY_STR	**HEX**:|AB|F5|	1042
Derisbi	malware	dfb81a	**HEX**:|FF|FF|FF|	**HEX**:|00|00|00|	5000
P1	benign	068164	**STR**:ANY_STR		4433,6881
RC2	benign	094bff	**STR**:ANY_STR		12345
GameServer	benign	10b681	**STR**:ANY_STR	**HEX**:|20|C1|	49363
WIService	benign	13c692	**STR**:ANY_STR		8888,9888
Sality	malware	18e8e9	**STR**:ANY_STR		5429
RAdmin	malware	1bed4d	**STR**:ANY_STR		5931
DroidCam	benign	2f4825	**STR**:ANY_STR	**STR**:CMD /v2	4747
Server	benign	3bc69a	**STR**:ANY_STR	**HEX**:|25|39|40	9711
BitComet	malware	5256e4	**STR**:ANY_STR		11912
Hupigon	malware	63e712	**STR**:ANY_STR	**STR**:HTTP/1.0	3838
ddegw	benign	96d3e9	**STR**:ANY_STR	**STR**:ZLSGW	5000
PfPk!1g	malware	c9b147	**STR**:ANY_STR	**HEX**:|FD|FD|00|	9910
AeroAdmin	benign	e9bac5	**STR**:ANY_STR		5950

Class†: classification result from VirusTotal,

Table 4.17: Extracted network scanning signatures. We ran BLUEPRINT and extracted various scanning signatures from the generated harness (*e.g.*, connected and scrap the banner) or the symbolic execution. "Payload" is sending data of the scanner and "Response" is the expected output to validate the victim.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis proposes two systems to strengthen and weaken the binary analysis process. For weakening the two notable binary analysis techniques, fuzzing and symbolic execution, this thesis proposes a new attack mitigation system, called FUZZIFICATION, for developers to prevent adversarial fuzzing. We develop three principled ways to hinder fuzzing: injecting delays to slow fuzzed executions; inserting fabricated branches to confuse coverage feedback; transforming data-flows to prevent taint analysis and utilizing complicated constraints to cripple symbolic execution. We design robust anti-fuzzing primitives to hinder attackers from bypassing FUZZIFICATION.

On the other hand, for enabling the fuzzing on the Windows application, this thesis presents WINNIE, an end-to-end system to support fuzzing Windows applications. Instead of repeatedly running the program directly, WINNIE synthesizes lightweight harnesses to directly invoke interesting functions, bypassing GUI code. It also features an implementation of `fork` on Windows to clone processes efficiently.

5.2 Future work

This dissertation presents three prototype systems: FUZZIFICATION, WINNIE, and BLUEPRINT. In this section, we discuss research topics to be handled in the near future.

5.2.1 Delay primitive on different H/W environments

We adopt CSmith-generated code as our delay primitives using measured delay on one machine (*i.e.*, developer's machine). This configuration implies that those injected delays

might not be able to bring the expected slow down to the fuzzed execution with more powerful hardware support. On the other hand, the delay primitives can cause higher overhead than expected for regular users with less powerful devices. To handle this, we plan to develop an additional variation that can dynamically adjust the delay primitives at runtime. Specifically, we measure the CPU performance by monitoring a few instructions and automatically adjusting a loop counter in the delay primitives to realize the accurate delay in different hardware environments. However, the code may expose static pattern such as time measurement system call or a special instruction like `rdtsc`; thus we note that this variation has inevitable trade-off between adaptability and robustness.

5.2.2 Handling complicated data structure in harness

Custom structures. Beyond this initial work towards practical Windows fuzzing, we identify several directions for future improvement. Among the following, we believe that handling structures and callback functions is fundamentally challenging, whereas supporting other ABIs or languages would be relatively straightforward.

Structures. Custom structures are challenging to both automatic testing tools and human researchers, and incorrect structures may lead to program crashes. To mitigate this issue, we could apply a memory pre-planning technique [135] to provide probabilistic guarantees to avoid crashes. We could also use memory breakpoints to trace the detailed memory access patterns of the program and infer the structure layouts.

Callback functions. Callback functions in the main executable make harness generation difficult. In our example Figure 4.3, we reconstructed the callback function by copying decompiled code from the main binary into the harness. For simple callbacks, we could automatically add decompiled code to the harness. For complicated cases, we could load the main binary and call the functions directly, as copied code is not always reliable.

Support for Non-C ABIs. WINNIE focuses on C-style APIs, and we did not investigate fuzzing programs with other ABIs. In our experience during the evaluation, these libraries

are rare in practice. In the future, WINNIE can be extended to support other native languages' ABIs, like C++, Rust, or Go.

Bytecode languages and interpreted binaries. While WINNIE supports most native applications, it does not support applications compiled for a virtual machine (*e.g.*, .NET, Java). To support these binaries, specialized instrumentation techniques [136] should be used to collect code coverage.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[2] M. Zalewski, *American Fuzzy Lop (2.52b)*, http://lcamtuf.coredump.cx/afl/, 2018.

[3] LLVM, *LibFuzzer - A Library for Coverage-guided Fuzz Testing*, http://llvm.org/docs/LibFuzzer.html, 2017.

[4] Google, *Syzkaller - Linux Syscall Fuzzer*, https://github.com/google/syzkaller, 2016.

[5] ——, *Honggfuzz*, https://google.github.io/honggfuzz/, 2016.

[6] M. Eddington, "Peach Fuzzing Platform," *Peach Fuzzer*, p. 34, 2011.

[7] CENSUS, *Choronzon - An Evolutionary Knowledge-based Fuzzer*, ZeroNights Conference, 2015.

[8] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek, "Hackers vs. Testers A Comparison of Software Vulnerability Discovery Processes," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[9] Synopsys, *Where the Zero-days are*, https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/state-of-fuzzing-2017.pdf, 2017.

[10] M. Hafiz and M. Fang, "Game of Detections: How Are Security Vulnerabilities Discovered in the Wild?" *Empirical Software Engineering*, vol. 21, no. 5, pp. 1920–1959, Oct. 2016.

[11] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Department of Computer Science, University of Auckland, New Zealand, Tech. Rep., 1997.

[12] ——, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, USA, 1998.

[13] M. Miller, *Trends, Challenges, And Strategic Shifts In The Software Vulnerability Mitigation Landscape*, https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends,Challenges,andStrategicShiftsintheSoftwareVulnerabilityMitigationLandscape.pdf, BlueHat IL, 2019.

[14] Emma Woollacott, *Windows Of Opportunity: Microsoft OS Remains The Most Lucrative Target For Hackers*, https://portswigger.net/daily-swig/windows-of-opportunity-microsoft-os-remains-the-most-lucrative-target-for-hackers, The Daily Swig, 2018.

[15] A. Fiscutean, *Microsoft Office Now The Most Targeted Platform, As Browser Security Improves*, https://www.csoonline.com/article/3390221/microsoft-office-now-the-most-targeted-platform-as-browser-security-improves.html, CSO, 2019.

[16] Brian Krebs, *The Scrap Value of a Hacked PC*, https://krebsonsecurity.com/2012/10/the-scrap-value-of-a-hacked-pc-revisited/, 2012.

[17] D. Palmer, *Top Ten Security Vulnerabilities Most Exploited By Hackers*, https://www.zdnet.com/article/these-are-the-top-ten-security-vulnerabilities-most-exploited-by-hackers-to-conduct-cyber-attacks/, ZDNet, 2019.

[18] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing through Selective Symbolic Execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[20] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments.," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[21] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[22] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[23] M. Rash, *A Collection of Vulnerabilities Discovered by the AFL Fuzzer*, https://github.com/mrash/afl-cve, 2017.

[24] Syzkaller, *Syzkaller Found Bugs - Linux Kernel*, https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[25] Google, *Honggfuzz Found Bugs*, https://github.com/google/honggfuzz#trophies, 2018.

[26] O. Chang, A. Arya, and J. Armour, *OSS-Fuzz: Five Months Later, and Rewarding Projects*, https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2018.

[27] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[28] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

[29] Google, *Fuzzing for Security*, https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[30] ——, *OSS-Fuzz - Continuous Fuzzing for Open Source Software*, https://github.com/google/oss-fuzz, 2016.

[31] Microsoft, *Microsoft Previews Project Springfield, a Cloud-based Bug Detector*, https://blogs.microsoft.com/next/2016/09/26/microsoft-previews-project-springfield-cloud-based-bug-detector, 2016.

[32] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

[33] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

[34] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[35] M. Zalewski, *American fuzzy lop*, http://lcamtuf.coredump.cx/afl/, 2015.

[36] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[37] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[39] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing," in *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.

[40] C. Lemieux and K. Sen, "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage," *ArXiv e-prints*, Sep. 2017.

[41] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[42] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[43] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[44] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[45] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, Carnegie Mellon University Pittsburgh, PA, 2012.

[46] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[47] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2006.

[48] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[49] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[50] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[51] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[52] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[53] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[54] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[55] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference (ESEC) / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sep. 2005.

[56] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[57] O. Whitehouse, *Introduction to Anti-Fuzzing: A Defence in Depth Aid*, https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/, 2014.

[58] D. Göransson and E. Edholm, "Escaping the Fuzz," Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[59] C. Miller, *Anti-Fuzzing*, https://www.scribd.com/document/316851783/anti-fuzzing-pdf, 2010.

[60] Z. Hu, Y. Hu, and B. Dolan-Gavitt, "Chaff Bugs: Deterring Attackers by Making Software Buggier," *CoRR*, vol. abs/1808.00659, 2018. arXiv: 1808.00659.

[61] K. Li, "Afl's blindspot and how to resist afl fuzzing for arbitrary elf binaries," *Black Hat USA Briefings (Black Hat USA), Las Vegas, NV*, 2018.

[62] UPX Team, *The Ultimate Packer for eXecutables*, https://upx.github.io, 2017.

[63] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – Software Protection for the Masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, B. Wyseur, Ed., IEEE, 2015.

[64] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[65] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault Tolerance during Execution," *Proceedings of the IEEE COMPSAC*, pp. 149–155, 1977.

[66] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela, "Software Diversity: State of the Art and Perspectives," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 14, no. 5, pp. 477–495, Oct. 2012.

[67] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, no. 2, pp. 220–232, 1975.

[68] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[69] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[70] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[71] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[72] M. Zalewski, *Fuzzing Random Programs without execve()*, https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014.

[73] ——, *New in AFL: Persistent Mode*, https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html, 2015.

[74] Z. Xu, *PTfuzzer*, https://github.com/hunter-ht-2018/ptfuzzer, 2018.

[75] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[76] M. Zalewski, *High-performance Binary-only Instrumentation For AFL-fuzz*, https://github.com/mirrorer/afl/tree/master/qemu_mode, 2016.

[77] C. online, *Seven of the Biggest Recent Hacks on Crypto Exchanges*, https://www.ccn.com/japans-16-licensed-cryptocurrency-exchanges-launch-self-regulatory-body/, 2018.

[78] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, Apr. 2005.

[79] T. G. Derek Bruening Vladimir Kiriansky, *Dynamic Instrumentation Tool Platform*, http://www.dynamorio.org/, 2009.

[80] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[81] *WinAFL Crashes with Testing Code*, https://github.com/ivanfratric/winafl/issues/62, 2017.

[82] *Unexplained Crashes in WinAFL*, https://github.com/DynamoRIO/dynamorio/issues/2904, 2018.

[83] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 283–294.

[84] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2007.

[85] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[86] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[87] GNU Project, *GNU Binutils Collection*, https://www.gnu.org/software/binutils, 1996.

[88] M. Zalewski, *Technical Whitepaper for AFL-fuzz*, https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt, 2017.

[89] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *Proceedings of the USENIX Annual Technical Conference*, 2002.

[90] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe Retrofitting of Legacy Code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[91] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[92] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-Sensitive Control Security," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[93] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[94] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[95] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[96] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring Deep State Spaces via Fuzzing," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[97] XnSoft, *XnView Image Viewer*, https://www.xnview.com/en/, 2020.

[98] AutoIt Consulting Ltd, *AutoIt Scripting Language*, https://www.autoitscript.com/site/autoit/, 2019.

[99] R. Freingruber, *Fuzzing Closed Source Applications*, https://def.camp/wp-content/uploads/dc2017/Day_1_Rene_Fuzzing_closed_source_applications_DefCamp.pdf, 2017.

[100] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "FUDGE: Fuzz Driver Generation At Scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2019, pp. 975–985.

[101] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic Fuzzer Generation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[102] Y. Alon and N. Ben-Simon, *50 CVEs In 50 Days: Fuzzing Adobe Reader*, https://research.checkpoint.com/50-adobe-cves-in-50-days/, 2018.

[103] R. Schaefer, *Fuzzing Adobe Reader For Exploitable Vulns*, https://kciredor.com/fuzzing-adobe-reader-for-exploitable-vulns-fun-not-profit.html, 2018.

[104] symeon, *Fuzzing The MSXML6 Library With WinAFL*, https://symeonp.github.io/2017/09/17/fuzzing-winafl.html, 2017.

[105] J. Min, *Using WinAFL To Fuzz Hangul(HWP) AppShield*, https://sigpwn.io/blog/2018/1/29/using-winafl-to-fuzz-hangul-appshield, 2018.

[106] R. Freingruber, *Hack The Hacker: Fuzzing Mimikatz On Windows With Winafl & Heatmaps*, https://sec-consult.com/en/blog/2017/09/hack-the-hacker-fuzzing-mimikatz-on-windows-with-winafl-heatmaps-0day/, 2017.

[107] M. Zalewski, *Fuzzing Random Programs Without Execve()*, https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2019.

[108] Google, *A New Chapter For OSS-Fuzz*, https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html, 2018.

[109] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing By Program Transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[110] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, *AFL For Fuzzing Windows Binaries*, https://github.com/ivanfratric/winafl, 2016.

[111] Google, *Honggfuzz*, https://github.com/google/honggfuzz, 2010.

[112] H. Gray, *Fuzzing Linux GUI/GTK Programs With American Fuzzy Lop (AFL) For Fun And Pr... You Get the Idea.* https://blog.hyperiongray.com/fuzzing-gtk-programs-with-american-fuzzy-lop-afl/.

[113] Google, *OSS-Fuzz - continuous fuzzing of open source software*, https://github.com/google/oss-fuzz, 2016.

[114] XnSoft, *Supported file formats in XnView*, https://www.xnview.com/en/xnviewmp/formats, 2019.

[115] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, *How to Select A Target Function*, https://github.com/googleprojectzero/winafl#how-to-select-a-target-function, 2016.

[116] I. Guilfanov, *IDA Pro - Hex Rays*, https://www.hex-rays.com/products/ida/, 2018.

[117] N. S. Agency, *Ghidra Software Reverse Engineering Framework*, https://ghidra-sre.org/, 2019.

[118] PaX Team, *PaX Address Space Layout Randomization (ASLR)*, http://pax.grsecurity.net/docs/aslr.txt, 2003.

[119] Dmytro Oleksiuk, *fork() for Windows*, https://gist.github.com/Cr4sh/126d844c28a7fbfd25c6, 2016.

[120] M. Russinovich and D. A. Solomon, *Windows internals: including Windows server 2008 and Windows Vista*. Microsoft press, 2009.

[121] C. authors, *Highlights of Cygwin Functionality*, https://cygwin.com/cygwin-ug-net/highlights.html, 1996.

[122] Microsoft, *Frequently Asked Questions about Windows Subsystem for Linux*, https://docs.microsoft.com/en-us/windows/wsl/faq, 2018.

[123] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with

dynamic instrumentation," in *Acm sigplan notices*, ACM, vol. 40, 2005, pp. 190–200.

[124] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, *WinAFL Intel PT mode*, https://github.com/googleprojectzero/winafl/blob/master/readme_pt.md, 2019.

[125] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead Through Coverage-guided Tracing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[126] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[127] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[128] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing.," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[129] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, "Automatic generation of string signatures for malware detection," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2009, pp. 101–120.

[130] J. Caballero, P. Poosankam, S. McCamant, D. Babi ć, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Oct. 2009.

[131] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.

[132] G. Sood, *virustotal: R Client for the virustotal API*, R package version 0.2.1, 2017.

[133] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, IEEE, 2017, pp. 8–9.

[134] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[135] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "Pmp: Cost-effective forced execution with probabilistic memory pre-planning," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[136] 0xd4d, *.NET module/assembly reader/writer library*, https://github.com/0xd4d/dnlib, 2013.